FARSIGHT



第 1 章　ARM 系列微处理器简介

1.1　什么是 ARM

ARM（Advanced RISC Machines）有三种含义，它是一个公司的名称，是一类微处理器的通称，还是一种技术的名称。

ARM 公司是微处理器行业的一家知名企业，设计了大量高性能、廉价、低耗能的 RISC（Reduced Instruction Set Computing，精简指令集计算机处理器）芯片，并开发了相关技术和软件。ARM 处理器具有性能高、成本低和能耗低的特点，适用于嵌入式控制、消费/教育类多媒体、DSP 和移动式应用等领域。

ARM 公司本身不生产芯片，靠转让设计许可，由合作伙伴公司来生产各具特色的芯片。ARM 这种商业模式的强大之处在于其价格合理，它在全世界范围的合作伙伴超过 100 个，其中包括许多著名的半导体公司。ARM 公司专注于设计，设计的芯片内核耗电少，成本低，功能强，特有 16/32 位双指令集。ARM 已成为移动通信、手持计算和多媒体数字消费等嵌入式解决方案的 RISC 实际标准。

1.1.1　ARM 公司历史

1990 年 11 月 ARM 公司成立于英国，原名 Advanced RISC Machine 有限公司，是苹果电脑、Acorn 电脑集团和 VLSI Technology 的合资企业。Acorn 曾推出世界首个商用单芯片 RISC 处理器，而苹果电脑当时希望将 RISC 技术应用于自身系统，ARM 微处理器新标准因此应运而生。ARM 公司成功地研制了首个低成本 RISC 架构，迅速在市场上崭露头角。与此同时，RISC 结构的竞争对手都着眼于提高性能，发展适合高端工作站处理器的 RISC 结构。

1991 年 ARM 公司推出首个嵌入式 RISC 核心——ARM6™系列处理器后不久，VLSI 率

先获得授权，一年后夏普和 GEC Plessey 也成为授权用户。1993 年德州仪器和 Cirrus Logic 也签署了授权协议。从此 ARM 公司的知识产权产品和授权用户都急剧增多。1993 年 Nippon Investment and Finance（NIF）成为 ARM 公司股东后，ARM 公司开始向全球拓展，分别在亚洲、北美洲和欧洲设立了办事处。1998 年 4 月 ARM 公司在伦敦证券交易所和纳斯达克交易所上市。

ARM 公司现已发展成为一家全球性大公司，公司在英国、法国和美国设有研发中心，在中国、法国、德国、日本、韩国、以色列、英国和美国建立了销售、行政和技术支持办事处。ARM 中国—安谋咨询上海有限公司于 2002 年 7 月成立。

1.1.2　ARM 的商业模式

ARM Holdings（伦敦证交所：ARM：纳斯达克：ARMHY）在半导体革新过程中初露峥嵘，被 Dataquest 誉为世界第一的知识产权供应商。20 世纪 90 年代初，ARM 公司率先推出 32 位 RISC 微处理器芯片系统（SoC）知识产权公开授权概念。ARM 公司通过出售芯片技术授权而非生产或销售芯片，建立起新型的微处理器设计、生产和销售商业模式。采用 ARM 技术的微处理器遍及各类电子产品，在汽车电子、消费娱乐、成像、工业控制、网络、存储安保和无线等领域 ARM 技术无处不在。

ARM 公司知识产权授权用户众多，其中包括世界顶级的半导体公司。全球 20 家最大的半导体厂商中有 19 家是 ARM 公司的用户。这些合作伙伴通过使用 ARM 公司低价、高效的 IP 核技术研制生产微处理器、外围设备和系统芯片。迄今这些厂商共发售了超过 10 亿个 ARM 微处理器内核。

为支持和增补 ARM 公司的现有 RISC 微处理器内核和 SoC IP，公司开发了功能强大的软件。ARM 公司的伙伴企业能够获得各种基于软件的 IP、操作系统端口和软件设计服务，从而大大降低产品开发风险，缩短上市时间。

首先是 ARM PrimeXsys 平台。这是一种取出即用的 IP，以平台的形式为专门应用提供支持。第一个 PrimeXsys 平台是 2001 年 9 月推出的 PrimeXsys Wireless 平台。它是一个高集成度的可扩展平台，包含了所有必需的硬件、软件和集成工具。ARM 公司的伙伴企业可以利用这个平台轻松开发一系列基于 ARM 处理器的面向应用的设备，既迅速，风险又低。

ARM 公司推出的另一新技术是 Jazelle，这项技术能将 Java 技术和全球领先的 32 位嵌入式 RISC 结构结合在一起，使平台开发人员能够在同一处理器上与现有操作系统、中间软件和应用编码同时运行 Java 应用程序，从而提高性能，降低系统成本，比协处理器和双处理器解决方案能耗更低。

1.2　ARM 体系结构的命名规则

ARM 体系结构是 CPU 产品所使用的一种体系结构，ARM 公司开发了一套拥有知识产

权的 RISC 体系结构的指令集。每个 ARM 处理器都有一个特定的指令集架构，而一个特定的指令集架构又可以由多种处理器实现。

特定的指令集架构随着嵌入式市场的发展而发展。由于所有产品均采用一个通用的软件体系，所以相同的软件可在所有产品中运行（理论上如此）。

ARM 产品通常以 ARM【x】【y】【z】【T】【D】【M】【I】【E】【J】【F】【-S】形式出现。表 1.1 显示了 ARM 体系结构的命名规则中这些后缀的具体含义。

表 1.1 ARM 体系结构的命名规则

后 缀 变 量	含 义
x	系列，号如 ARM7、ARM9
y	存储管理/保护单元
z	Cache
T	Thumb16 位译码器
D	JTAG 调试器
M	快速乘法器
I	嵌入式跟踪宏单元
E	增强指令（基于 TDMI）
J	Jazelle 加速
F	向量浮点单元
S	可综合版本

另外，还有一些附加的要点：

① ARM7TDMI 之后的所有 ARM 内核，即使“ARM”标志后没有包含“TDMI”字符，也都默认包含了 TDMI 的功能特性；

② JTAG 是由 IEEE 1149.1 标准测试访问端口和边界扫描结构来描述的，它是 ARM 用来发送和接收处理器内核与测试仪器之间调试信息的一系列协议；

③ 嵌入式 ICE 宏单元是建立在处理器内部用来设置断点和观察点的调试硬件；

④ 可综合，意味着处理器内核是以源代码形式提供的。这种源代码形式可被编译成一种易于 EDA 工具使用的形式。

1.3 初识 ARM 系列处理器

ARM 处理器当前有 6 个产品系列：ARM7、ARM9、ARM9E、ARM10E、ARM11 和 SecurCore，其中 ARM11 为最近推出的产品。进一步的产品来自于合作伙伴，例如 Intel Xscale ARM7、ARM9、ARM9E、ARM10E 是 4 个通用处理器系列。每个系列提供一套特定的性能来满足设计者对功耗、性能、体积的需求。SecurCore 是第 5 个产品系列，是专门为安全设备而设计的。

表 1.2 总结了 ARM 各系列处理器所包含的不同类型。

表 1.2 ARM 各系列处理器所包含的不同类型

ARM 系列	包 含 类 型
ARM7 系列	ARM7EJ-S ARM7TDMI ARM7TDMI-S ARM720T
ARM9/9E 系列	ARM920T

续表

ARM 系列	包 含 类 型
ARM9/9E 系列	ARM922T ARM926EJ-S ARM940T ARM946E-S ARM966E-S ARM968E-S
向量浮点运算（Vector Floating Point）系列	VFP9-S VFP10
ARM10E 系列	ARM1020E ARM1022E ARM1026EJ-S
ARM11 系列	ARM1136J-S ARM1136JF-S ARM1156T2(F)-S ARM1176JZ(F)-S ARM11 MPCore
SecurCore 系列	SC100 SC110 SC200 SC210
其他合作伙伴产品	StrongARM XScale Cortex-M3 MBX

本节简要介绍 ARM 各个系列处理器的特点。

1.3.1 ARM7 系列

ARM7 内核采用冯·诺伊曼体系结构，数据和指令使用同一条总线。内核有一条 3 级流水线，执行 ARMv4 指令集。

ARM7 系列处理器主要用于对功耗和成本要求比较苛刻的消费类产品。其最高主频可以

到达 130MIPS（MIPS 指每秒执行的百万条指令数）。ARM7 系列包括 ARM7TDMI、ARM7TDMI-S、ARM7EJ-S 和 ARM720T 4 种类型，主要用于适应不同的市场需求。

ARM7 系列处理器主要具有以下特点：

- 成熟的大批量的 32 位 RICS 芯片；
- 最高主频到达 130MIPS；
- 功耗低；
- 代码密度高，兼容 16 位微处理器；
- 开发工具多、EDA 仿真模型多；
- 调试机制完善；
- 提供 0.25μm、0.18μm 及 0.13μm 的生产工艺；
- 代码与 ARM9 系列、ARM9E 系列以及 ARM10E 系列兼容。

1.3.2 ARM9 系列

ARM9 系列于 1997 年问世。由于采用了 5 级指令流水线，ARM9 处理器能够运行在比 ARM7 更高的时钟频率上，改善了处理器的整体性能；存储器系统根据哈佛体系结构（程序和数据空间独立的体系结构）重新设计，区分了数据总线和指令总线。

ARM9 系列的第一个处理器是 ARM920T，包含独立的数据指令 Cache 和 MMU。此处理器能够被用在要求有虚拟存储器支持的操作系统上。此系列的 ARM922T 是 ARM920T 的变种，只有一半大小的数据指令 Cache。

ARM940T 包含一个更小的数据指令 Cache 和一个 MPU。它是针对不要求运行操作系统的应用而设计的。ARM920T、ARM940T 都执行 v4T 架构指令。

1.3.3 ARM9E 系列

ARM9 系列的下一个处理器是基于 ARM9E-S 内核的。这个内核是 ARM9 内核带有 E 扩展的一个可综合版本。它有 ARM946E-S 和 ARM966E-S 两个变种。两者都执行 v5TE 架构指令。它们也支持可选的嵌入式跟踪宏单元，支持开发者实时跟踪处理器上指令和数据的执行。当调试对时间敏感的程序段时，这种方法非常重要。

ARM946E-S 包括 TCM、Cache 和一个 MPU。TCM 和 Cache 的大小可配置。该处理器是针对要求有确定的实时响应的嵌入式控制而设计的。ARM966E-S 有可配置的 TCM，但没有 MPU 和 Cache 扩展。

ARM9 系列的 ARM926EJ-S 内核为可综合的处理器内核，发布于 2000 年。它是针对小型便携式 Java 设备，诸如 3G 手机和 PDA 应用而设计的。ARM926EJ-S 是第一个包含 Jazelle 技术，可加速 Java 字节码执行的 ARM 处理器内核。它还有一个 MMU、可配置的 TCM 以及具有零或非零等待存储器的数据/指令 Cache。

1.3.4 ARM10 系列

ARM10 发布于 1999 年，具有高性能、低功耗的特点。它所采用的新的体系使其在所有

ARM 产品中具有最高的 MIPS/MHz。它将 ARM9 的流水线扩展到 6 级，也支持可选的向量浮点单元 VFP，对 ARM10 的流水线加入了第 7 段。VFP 明显增强了浮点运算性能并与 IEEE 754.1985 浮点标准兼容。

ARM10E 系列处理器采用了新的节能模式，提供了 64 位的 Load/Store 体系，支持包括向量操作的满足 IEEE 754 的浮点运算协处理器，系统集成更加方便，拥有完整的硬件和软件开发工具。ARM10E 系列包括 ARM1020E、ARM1022E 和 ARM1026EJ-S 3 种类型。

1.3.5　ARM11 系列

ARM1136J-S 发布于 2003 年，是针对高性能和高能效应而设计的。ARM1136J-S 是第一个执行 ARMv6 架构指令的处理器。它集成了一条具有独立的 Load/Stroe 和算术流水线的 8 级流水线。ARMv6 指令包含了针对媒体处理的单指令流多数据流扩展，采用特殊的设计改善视频处理能力。

1.3.6　SecurCore 系列

SecurCore 系列处理器提供了基于高性能的 32 位 RISC 技术的安全解决方案。SecurCore 系列处理器除了具有体积小、功耗低、代码密度高等特点外，还具有它自己特别优势，即提供了安全解决方案支持。下面总结了 SecurCore 系列的主要特点：

① 支持 ARM 指令集和 Thumb 指令集，以提高代码密度和系统性能；
② 采用软内核技术以提供最大限度的灵活性，可以防止外部对其进行扫描探测；
③ 提供了安全特性，可以抵制攻击；
④ 提供面向智能卡和低成本的存储保护单元 MPU；
⑤ 可以集成用户自己的安全特性和其他的协处理器。

SecurCore 系列包含 SC100、SC110、SC200 和 SC210 4 种类型。

1.3.7　其他系列处理器

StrongARM 处理器最初是 ARM 公司与 Digital Semiconductor 公司合作开发的，现在由 Intel 公司单独许可。在低功耗、高性能的产品中应用很广泛。它是哈佛架构的，具有独立的数据和指令 Cache，有 MMU（Memory Management Unit）。StrongARM 是第一个包含 5 级流水线的高性能 ARM 处理器，但它不支持 Thumb 指令集。

Intel 公司的 Xscale 是 Strong ARM 的后续产品，在性能上有显著改善。它执行 v5TE 架构指令，也是哈佛结构的，类似于 StrongARM 也包含一个 MMU。

1.3.8　Cortex - M3 和 MPCore

为了适应市场的需要，ARM 推出了两个新的处理器：Cortex-M3 和 MPCore。Cortex-M3 主要针对微控制器市场，而 MPCore 主要针对高端消费类产品。

Cortex-M3 改进了代码密度，减少了中断延时并有更低的功耗。Cortex-M3 中实现了最新

了 Thumb-2 指令集。MPCore 提供了 Cache 一致性，每个支持 1～4 个 ARM11 核，这种设计为现代消费类产品对性能和功耗的需求作了很好的平衡。ARM 还引入了 L2Cache 控制器来改进系统的整体性能。

1.4 ARM 系列处理器的应用领域

1.4.1 ARM7 系列

ARM7 系列处理器主要应用于下面一些场合：

- 个人音频设备（MP3 播放器、WMA 播放器、AAC 播放器）；
- 接入级的无线设备；
- 喷墨打印机；
- 数码照相机；
- PDA。

1.4.2 ARM9 系列

ARM9 系列处理器具体应用于下面一些场合：

- 下一代无线设备，包括视频电话和 PDA 等；
- 数字消费品，包括机顶盒、家庭网关、MP3 播放器和 MPEG4 播放器；
- 成像设备，包括打印机、数码照相机和数码摄像机；
- 汽车、通信和信息系统。

1.4.3 ARM9E 系列

ARM9E 系列处理器具体应用于下面一些场合：

- 下一代无线设备，包括视频电话和 PDA 等；
- 数字消费品，包括机顶盒、家庭网关、MP3 播放器和 MPEG4 播放器；
- 成像设备，包括打印机、数码照相机和数码摄像机；
- 存储设备，包括 DVD 或 HDD 等；
- 工业控制，包括电机控制等；
- 汽车、通信和信息系统的 ABS 和车体控制；
- 网络设备，包括 VoIP、WirelessLAN 和 xDSL 等。

1.4.4 ARM10E 系列

ARM10E 系列处理器具体应用于下面一些场合：

- 下一代无线设备，包括视频电话和 PDA、笔记本电脑和互联网设备；

- 数字消费品，包括机顶盒、家庭网关、MP3 播放器和 MPEG4 播放器；
- 成像设备，包括打印机、数码照相机和数码摄像机；
- 汽车、通信和信息系统等；
- 工业控制，包括马达控制等。

1.4.5　SecureCore 系列

SecureCore 系列处理器主要应用于一些安全产品及应用系统，包括电子商务、电子银行业务、网络、移动媒体和认证系统等。

1.5　ARM 芯片的特点与选型

1.5.1　不同系列处理器间的比较

表 1.3 显示了 ARM7、ARM9、ARM10 及 ARM11 内核之间属性的比较。有些属性依赖于生产过程和工艺，具体芯片需参阅其芯片手册。

表 1.3　　ARM 系列处理器属性比较

项　　目	ARM7	ARM9	ARM10	ARM11
流水线深度	3 级	5 级	6 级	8 级
典型频率（MHz）	80	150	260	335
功耗（mw/ MHz）	0.06	0.19（+Cache）	0.5（+Cache）	0.4（+Cache）
MIPS/ MHz	0.97	1.1	1.3	1.2
架构	冯•诺伊曼	哈佛	哈佛	哈佛
乘法器	8×32	8×32	16×32	16×32

表 1.4 总结了各种处理器的不同功能。

表 1.4　　ARM 处理器不同功能特性

CPU 核	MMU/MPU	Cache	Jazelle	Thumb	指令集	E
ARM7TDMI	无	无	否	是	v4T	否
ARM7EJ-S	无	无	是	是	v5TEJ	是
ARM720T	MMU	统一 8KBCache	否	是	v4T	否
ARM920T	MMU	独立 16KB 指令和数据 Cache	否	是	v4T	否
ARM922T	MMU	独立 8KB 指令和数据 Cache	否	是	v4T	否
ARM926EJ-S	MMU	Cache 和 TCM 可配置	是	是	v5TEJ	是
ARM940T	MPU	独立 4KB 指令和数据 Cache	否	是	v4T	否
ARM946E-S	MPU	Cache 和 TCM 可配置	否	是	v5TE	是

ARM966E-S	无	Cache 和 TCM 可配置	否	是	v5TE	是
ARM1020E	MMU	独立 32KB 指令和数据 Cache	否	是	v5TE	是
ARM1022E	MMU	独立 16KB 指令和数据 Cache	否	是	v5TE	是
ARM1026EJ-S	MMU	Cache 和 TCM 可配置	是	是	v5TE	是
ARM1036J-S	MMU	Cache 和 TCM 可配置	是	是	v6	是
ARM1136JF-S	MMU	Cache 和 TCM 可配置	是	是	v6	是

1.5.2 ARM 芯片的选型

随着国内嵌入式应用领域的发展，ARM 芯片必然会获得广泛的重视和应用。但是由于 ARM 芯片有多达十几种的芯核结构、70 多芯片生产厂家以及千变万化的内部功能配置组合，开发人员在选择方案时会有一定的困难。所以对 ARM 芯片做对比研究是十分必要的。

1．ARM 芯片选择的一般原则

从应用角度看，在选择 ARM 芯片时应从以下几个方面考虑。

（1）ARM 芯核

如果希望使用 Windows CE 或 Linux 等操作系统以减少软件开发时间，就需要选择 ARM720T 以上带有 MMU（Memory Management Unit）功能的 ARM 芯片，ARM720T、StrongARM、ARM920T、ARM922T、ARM946T 都带有 MMU 功能。而 ARM7TDMI 没有 MMU，不支持 Windows CE 和大部分的 Linux；但目前有 uCLinux 等少数几种 Linux 不需要 MMU 的支持。

（2）系统时钟控制器

系统时钟决定了 ARM 芯片的处理速度。ARM7 的处理速度为 0.97MIPS/MHz，常见的 ARM7 芯片系统主时钟为 20～133MHz，ARM9 的处理速度为 1.1MIPS/MHz，常见的 ARM9 的系统主时钟为 100～233MHz，ARM10 最高可以达到 700MHz。不同芯片对时钟的处理不同，有的芯片只有一个主时钟频率，这样的芯片可能不能同时顾及 UART 和音频时钟准确性，如 Cirrus Logic 的 EP7312 等；有的芯片内部时钟控制器可以分别为 CPU 核和 USB、UART、DSP、音频等功能部件提供同频率的时钟，如 PHILIPS 公司 SAA7750 等芯片。

（3）内部存储器容量

在不需要大容量存储器时，可以考虑选用有内置存储器的 ARM 芯片。表 1.5 列出了内置存储器的 ARM 芯片。

表 1.5 内置存储器的 ARM 芯片

芯 片 型 号	供 应 商	Flash 容量	ROM 容量	SDAM 容量
AT91F40162	ATMEL	2MB	256KB	4KB
AT91FR4081	ATMEL	1MB		128KB
SAA7750	Philips	384KB		64KB
PUC3030A	Micornas	256KB		56KB

HMS30C7272	Hynix	192KB		
LC67F500	Snayo	640KB		32KB

（4）USB 接口

许多 ARM 芯片内置有 USB 控制器，有些芯片甚至同时有 USB Host 和 USB Slave 控制器。表 1.6 显示了内置 USB 控制器的 ARM 芯片。

表 1.6　　内置 USB 控制器的 ARM 芯片

芯 片 型 号	ARM 内核	供　应　商	USB Slave	USB Host	IIS 接口
S3C2410	ARM920T	Samsung	1	2	1
S3C2400	ARM920T	Samsung	1	2	1
S5N8946	ARM7TDMI	Samsung	1	0	0
L7205	ARM720T	Linkup	1	1	0
L7210	ARM720T	Linkup	1	1	0
EP9312	ARM920T	Cirrus logic	0	3	1
Dragonball MX1	ARM920T	Motorola	1	0	1
SAA7750	ARM720T	Plilips	1	0	1
TMS320DSC2x	ARM7TDMI	TI	1	0	0
PUC3030A	ARM7TDMI	Micronas	1	0	5
ML67100	ARM7TDMI	OKI	1	0	0
ML7051LA	ARM7TDMI	OKI	1	0	0
SA-1100	StrongARM	Intel	1	0	0

续表

芯 片 型 号	ARM 内核	供　应　商	USB Slave	USB Host	IIS 接口
LH7979531	ARM7TDMI	Sharp	1	0	0
GMS320C7201	ARM720T	Hynix	1	0	1

（5）GPIO 数量

在某些芯片供应商提供的说明书中，往往申明的是最大可能的 GPIO 数量，但是有许多引脚是和地址线、数据线、串口线等引脚复用的。这样在系统设计时需要计算实际可以使用的 GPIO 数量。

（6）中断控制器

ARM 内核只提供快速中断（FIQ）和标准中断（IRQ）两个中断向量。但各个半导体厂家在设计芯片时加入了自己定义的中断控制器，以便支持诸如串行口、外部中断、时钟中断等硬件中断。外部中断控制是选择芯片必须考虑的重要因素，合理的外部中断设计可以很大程度地减少任务调度工作量。例如 PHILIPS 公司的 SAA7750，所有 GPIO 都可以设置成 FIQ 或 IRQ，并且可以选择上升沿、下降沿、高电平和低电平 4 种中断方式。这使得红外线遥控接收、指轮盘和键盘等任务都可以作为背景程序运行。而 Cirrus Logic 公司的 EP7312 芯片只有 4 个外部中断源，并且每个中断源都只能是低电平或高电平中断，这样在接收红外线信号

的场合必须用查询方式，浪费大量 CPU 时间。

（7）IIS（Integrate Interface of Sound）接口

即集成音频接口。如果设计音频应用产品，IIS 总线接口是必需的。

（8）nWAIT 信号

这是一个外部总线速度控制信号。不是每个 ARM 芯片都提供这个信号引脚，利用这个信号与廉价的 GAL 芯片就可以实现与符合 PCMCIA 标准的 WLAN 卡和 Bluetooth 卡的接口，而不需要外加高成本的 PCMCIA 专用控制芯片。另外，当需要扩展外部 DSP 协处理器时，此信号也是必需的。

（9）RTC（Real Time Clock）

很多 ARM 芯片都提供实时时钟功能，但方式不同。如 Cirrus Logic 公司的 EP7312 的 RTC 只是一个 32 位计数器，需要通过软件计算出年月日时分秒；而 SAA7750 和 S3C2410 等芯片的 RTC 直接提供年月日时分秒格式。

（10）LCD 控制器

有些 ARM 芯片内置 LCD 控制器，有的甚至内置 64KB 彩色 TFT LCD 控制器。在设计 PDA 和手持式显示记录设备时，选用内置 LCD 控制器的 ARM 芯片（如 S3C2410）较为适宜。

（11）PWM 输出

有些 ARM 芯片有 2～8 路 PWM 输出，可以用于电机控制或语音输出等场合。

（12）ADC 和 DAC

有些 ARM 芯片内置 2～8 通道 8～12 位通用 ADC，可以用于电池检测、触摸屏和温度监测等。PHILIPS 的 SAA7750 更是内置了一个 16 位立体声音频 ADC 和 DAC，并且带耳机驱动。

（13）扩展总线

大部分 ARM 芯片具有外部 SDRAM 和 SRAM 扩展接口，不同的 ARM 芯片可以扩展的芯片数量即片选线数量不同，外部数据总线有 8 位、16 位或 32 位。为某些特殊应用设计的 ARM 芯片（如德国 Micronas 的 PUC3030A）没有外部扩展功能。

（14）UART 和 IrDA

几乎所有的 ARM 芯片都具有 1～2 个 UART 接口，可以用于和 PC 机通信或用 Angel 进行调试。一般的 ARM 芯片通信波特率为 115200bit/s，少数专为蓝牙技术应用设计的 ARM 芯片的 UART 通信波特率可以达到 920kbit/s，如 Linkup 公司 L7205。

（15）DSP 协处理器

表 1.7 总结了 ARM+DSP 结构的 ARM 芯片。

表 1.7 ARM+DSP 结构的 ARM 芯片

芯 片 型 号	供 应 商	DSP Core	DSP MIPS	应 用
TMS320DSC2X	TI	16bit C5000	500	数码照相机
Dragonball MX1	Motorola	24bit 56000		MP3 播放器
SAA7750	Philips	24bit EPIC	73	CD-MP3

VWS22100	Philips	16bit OAK		GSM
STLC1502	ST	D950	52	VoIP
GMS30C3201	Hynix	16bit Piccolo		
AT75C220	ATMEL	16bit OAK	40	
AT75C310	ATMEL	16bit OAK		
AT75C320	ATMEL	16bit OAK	40×2	
L7205	Linkup	16bit Piccolo	60×2	无线应用
L7210	Linkup	16bit Piccolo		
Quatro	OAK	16bit OAK		

（16）内置 FPGA

有些 ARM 芯片内置有 FPGA，适合于通信等领域。表 1.8 总结了 ARM+FPGA 结构的 ARM 芯片。

表 1.8　　ARM+FPGA 结构的 ARM 芯片

芯 片 型 号	供　应　商	ARM 芯核	FPGA 门数	引　脚　数
EPXA1	Altera	ARM922T	100000	484
EPXA4	Altera	ARM922T	400000	672
EPXA10	Altera	ARM922T	1000000	1020
TA7S20 系列	Triscend	ARM7TDMI	多种	多种

（17）时钟计数器和看门狗

一般 ARM 芯片都具有 2～4 个 16 位或 32 位时钟计数器和一个看门狗计数器。

（18）电源管理功能

ARM 芯片的耗电量与工作频率成正比，一般 ARM 芯片都有低功耗模式、睡眠模式和关闭模式。

（19）DMA 控制器

有些 ARM 芯片内部集成有 DMA（Direct Memory Access）接口，可以和硬盘等外部设备高速交换数据，同时减少数据交换时对 CPU 资源的占用。

另外，还可以选择的内部功能部件有：HDLC、SDLC、CD-ROM Decoder、Ethernet MAC、VGA controller、DC-DC。可以选择的内置接口有：IIC、SPDIF、CAN、SPI、PCI、PCMCIA。

最后需说明的是封装问题。ARM 芯片现在主要的封装有 QFP、TQFP、PQFP、LQFP、BGA、LBGA 等形式，BGA 封装具有芯片面积小的特点，可以减少 PCB 板的面积，但是需要专用的焊接设备，无法手工焊接。另外一般 BGA 封装的 ARM 芯片无法用双面板完成 PCB 布线，需要多层 PCB 板布线。

2．多芯核结构 ARM 芯片的选择

为了增强多任务处理能力、数学运算能力、多媒体以及网络处理能力，某些供应商提供

的 ARM 芯片内置多个芯核，目前常见的 ARM+DSP，ARM+FPGA，ARM+ARM 等结构。

（1）多 ARM 芯核

为了增强多任务处理能力和多媒体处理能力，某些 ARM 芯片内置多个 ARM 芯核。例如 Portal player 公司的 PP5002 内部集成了两个 ARM7TDMI 芯核，可以应用于便携式 MP3 播放器的编码器或解码器。从科胜讯公司（Conexant）分离出来的专门致力于高速通信芯片设计生产的 MinSpeed 公司在其多款高速通信芯片中集成了 2～4 个 ARM7TDMI 内核。

（2）ARM 芯核+DSP 芯核

为了增强数学运算功能和多媒体处理功能，许多供应商在其 ARM 芯片内增加了 DSP 协处理器。通常加入的 DSP 芯核有 ARM 公司的 Piccolo DSP 芯核、OAK 公司 16 位定点 DSP 芯核、TI 的 TMS320C5000 系列 DSP 芯核和 Motorola 的 56K DSP 芯核等。见表 1.7。

（3）ARM 芯核+FPGA

为了提高系统硬件的在线升级能力，某些公司在 ARM 芯片内部集成了 FPGA。见表 1.8。

3．选择方案举例

表 1.9 列举的最佳方案仅供参考，由于 SoC 集成电路的发展非常迅速，今天的最佳方案到明天就可能不是最佳的了。因此任何时候在选择方案时，都应广泛搜寻一下主要的 ARM 芯片供应商，以找出最适合的芯片。

表 1.9　　最佳应用方案推荐

应　用	第 一 方 案	第 二 方 案	备　注
高档 PDA	S3C2410	Dragon ball MX1	
便携 CD MP3 播放器	SAA7750		USB 和 CDROM 解码器
FLASH MP3 播放器	SAA7750	PUC3030A	内置 USB 和 FLASH
WLAN 和 BT 应用产品	L7205，L7210	Dragon ball MX1	高速串口和 PCMCIA 接口
VoiceOver IP	STLC1502		
数码照相机	TMS320DSC24	TMS320DSC21	内置高速图像处理 DSP

续表

应　用	第 一 方 案	第 二 方 案	备　注
便携式语音 email 机	AT75C320	AT75C310	内置双 DSP，可以分别处理 MODEM 和语音
GSM 手机	VWS22100	AD20MSP430	专为 GSM 手机开发
ADSL Modem	S5N8946	MTK－20141	
电视机顶盒	GMS30C3201		VGA 控制器
3G 移动电话机	MSM6000	OMAP1510	
10G 光纤通信	MinSpeed 公司系列 ARM 芯片		多 ARM 核+DSP 核

1.6 ARM 开发工具

用户选用 ARM 处理器开发嵌入式产品时，选择合适的开发工具可以加快开发进度，节省开发成本。根据功能不同，ARM 应用软件的开发工具分别有编译软件、汇编软件、连接软件、调试软件、评估板、JTAG 仿真器和在线仿真器等，目前世界上大约有四十多家公司提供以上不同种类的开发产品。

Realview 系列开发工具的英文全称为 Realview Developer Suite，是 ARM 公司（www.arm.com）为方便用户在 ARM 芯片上进行应用软件开发而推出的一整套集成开发工具。该套工具包括软件开发套件和硬件仿真工具。经过 ARM 公司逐年的维护和更新，目前的最新版本为 3.0。

ARM RVDS 起源于 ARM ADS（ARM Developer Suite），它对一些 ADS 的模块进行了增强并替换了一些 ADS 的组成部分。它支持几乎所有的 ARM 处理器，包括最新的 ARMv6 体系结构。支持的操作系统除了 Windows 外，还有 Linux。

ARM RVDS 主要包括以下几部分。

1．Realview compilation Tools

Realview compilation Tools 由编译器、汇编器和连接器组成。ARM 公司针对 ARM 系列每一种结构都进行了专门的优化和处理，这一点除了作为 ARM 结构的设计者 ARM 公司外，其他公司都无法办到。

Realview compilation Tools 主要包括以下组件：

- ARM/Thumb 汇编器 armasm；
- 连接器 armlink；
- 格式转换工具 fromelf；
- 库管理器 armar；
- C 和 C++应用程序库；
- 工程管理。

这些工具的使用过程如图 1.1 所示。

以上工具为命令行开发工具，同时也被集成在它的 IDE 开发环境中。

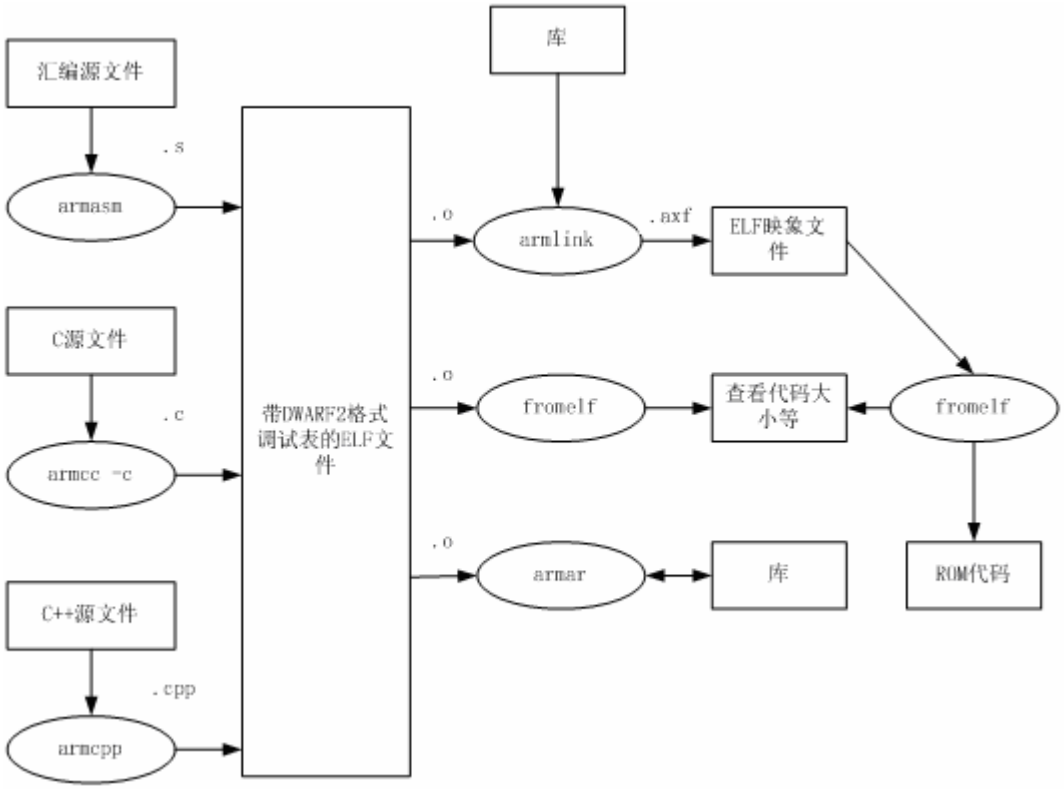


图 1.1 ARM 开发工具组件使用过程

2．集成开发环境

（1）CodeWarrior

CodeWarrior 是 Metrowerks 公司一套比较著名的集成开发环境，是一个直观、易用的环境，集成了很多 ARM 开发工具。CodeWarrior 界面风格独特，如图 1.2 所示。

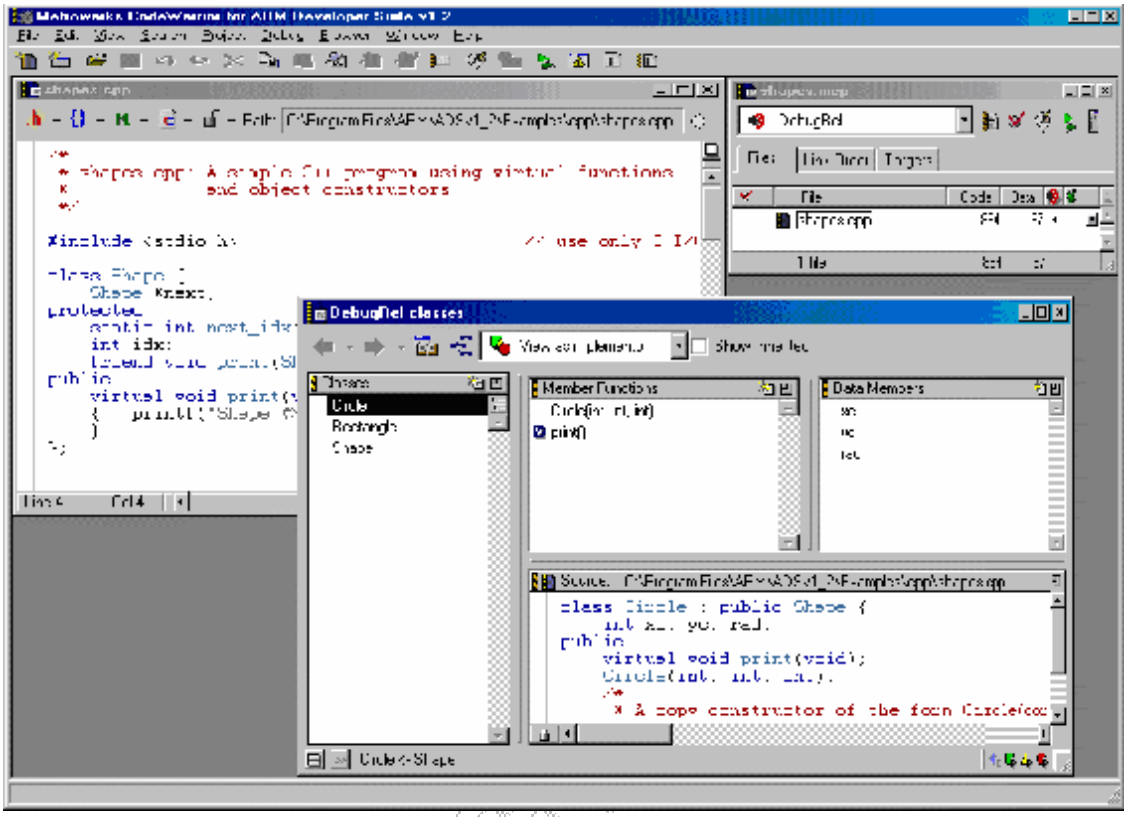


图 1.2　CodeWarrior 集成开发环境

CodeWarrior 包含项目管理、代码生成、语法敏感编辑器、C/C++源文件浏览器、类浏览器以及文件比较器等。项目管理有直观的 GUI，可以通过隐藏底层目录结构来简单地管理复杂的项目。强大的内置编辑器是编写软件的理想工具。可配置的接口让用户可以根据喜好裁减外形，以提高效率。

（2）AXD

AXD 即 ARM 扩展调试器（ARM extended Debugger）是运行在主机上的嵌入式开发调试工具。其界面如图 1.3 所示。

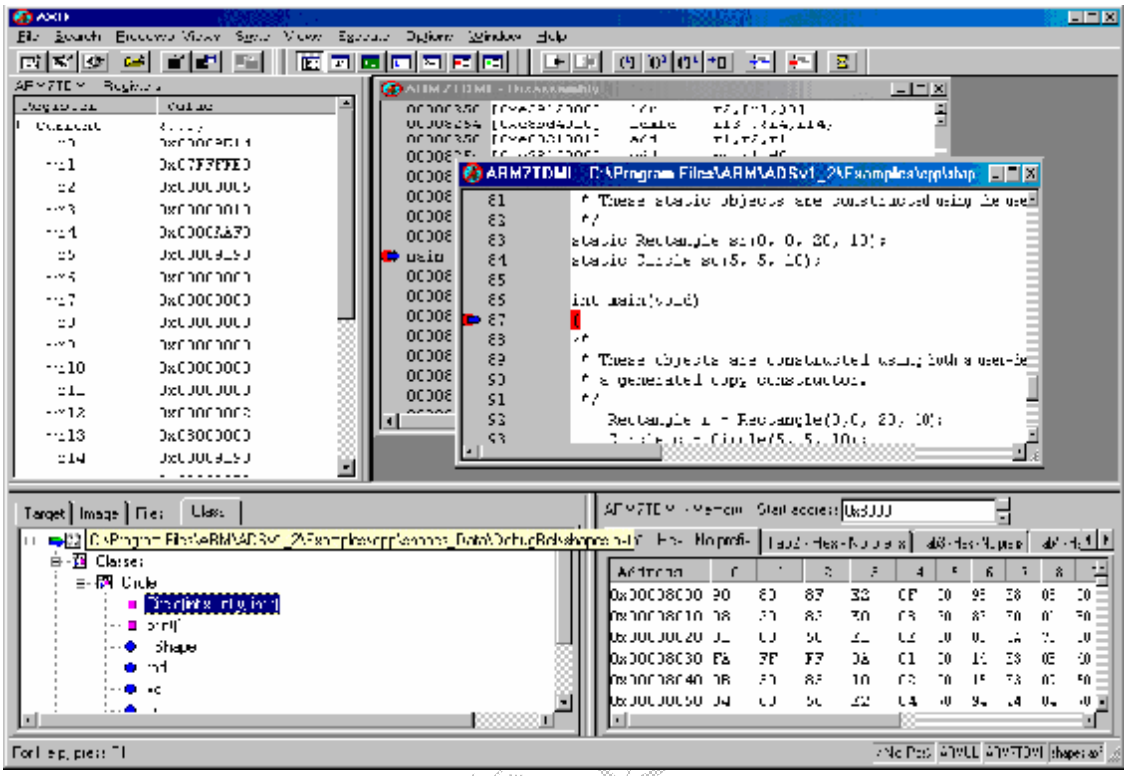


图 1.3 AXD 图形界面

AXD 包含新型的 GUI、图形窗口管理、数据显示、命令行接口等组件。它使用户不用改变调试器就可以选择不同的调试目标，如 ARMulator、Angel 或 Multi_ICE 等，扩展了 ARM 调试目标接口。

3．Multi_ICE

Multi_ICE 是 ARM 公司自己的 JTAG 仿真器，其 JTAG 链时钟可以设置为 5kHz～10MHz。它支持 ARM7、ARM9、ARM9E、ARM10 等 ARM 系列处理器。

Multi_ICE 主要有以下特点。

- 快速的下载和单步速度；
- 用户控制的输入、输出位；
- 可编程的 JTAG 位传送速率；
- 开放的接口，允许调试非 ARM 核和 DSP；
- 网络连接到多个调试器。

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 2 章 ARM 体系结构

华清远见

2.1 ARM 体系结构的特点

ARM 内核采用精简指令集结构（RISC，Reduced Instruction Set Computer）体系结构。RISC 技术产生于上世纪 70 年代。其目标是设计出一套能在高时钟频率下单周期执行、简单而有效的指令集，RISC 的设计重点在于降低硬件执行指令的复杂度，这是因为软件比硬件容易提供更大的灵活性和更高的智能。与其相对的传统复杂指令级计算机（CISC）则更侧重于硬件执行指令的功能性，使 CISC 指令变得更复杂。

RISC 的设计思想主要有以下特性。

- Load/Store 体系结构。

Load/Store 体系结构也称为寄存器/寄存器体系结构或者 RR 系统结构。在这类机器中，操作数和运算结果不是通过主存储器直接取回而是借用大量标量和矢量寄存器来取回的。与 RR 体系结构相反，还有一种存储器/存储器体系结构，在这种体系结构中，源操作数的中间值和最后的运算结果是直接从主存储器中取回的。这类机器的缩写符号是 SS 体系结构。

- 固定长度指令。

固定长度指令使得机器译码变得比较容易。由于指令简单，需要更多的指令来完成相同的工作，但是随着存储器存取速度的提高，处理器可以更快地执行较大代码段（即大量指令）。

- 硬联控制。

RISC 机以硬联控制指令为特点，而 CISC 的微代码指令则相反。使用 CISC（常常是可变长度的）指令集时处理器的语义效率最大，而简单指令往往容易被机器翻译。像 CISC 那样通过执行较少指令来完成工作未必省时，因为还要包括微代码译码所需要的时间。因此，由硬件实现指令在执行时间方面提供了更好的平衡。除此之外，还节省了芯片上用于存储微代码的空间并且消除了翻译微代码所需的时间。

- 流水线。

指令的处理过程被拆分为几个更小的、能够被流水线并行执行的单元。在理想情况下，流水线每周期前进一步，可获得更高的吞吐率。

- 寄存器。

RICS 处理器拥有更多的通用寄存器，每个寄存器都可存放数据或地址。寄存器可为所有的数据操作提供快速的局部存储访问。

表 2.1 总结了 RISC 和 CISC 之间主要的区别。

表 2.1 RISC 和 CISC 之间主要的区别

指　　标	RISC	CISC
指令集	一个周期执行一条指令，通过简单指令的组合实现复杂操作；指令长度固定	指令长度不固定，执行需要多个周期
流水线	流水线每周期前进一步	指令的执行需要调用微代码的一个微程序
寄存器	更多通用寄存器	用于特定目的的专用寄存器

Load/Store 结构	独立的 Load 和 Store 指令完成数据在寄存器和外部存储器之间的传输	处理器能够直接处理存储器中的数据

为了使 ARM 指令集能够更好地满足嵌入式应用的需要，ARM 指令集和单纯的 RISC 定义有以下几方面的不同。

- 一些特定指令的周期数可变

并非所有的 ARM 指令都是单周期的。例如，多寄存器转载/存储的 Load/Store 指令的周期数就不确定，必须根据被传送的寄存器个数而定。如果是访问连续的存储器地址，就可以改善性能，因为连续的存储器访问通常比随机访问要快。同时，代码密度也得到了提高，因为在函数的起始和结尾，多个寄存器的传输是很常用的操作。

- 内嵌桶形移位器产生更复杂的指令

内嵌桶形移位器是一个硬件部件，在一个输入寄存器被一条指令使用之前，内嵌桶形移位器可以处理该寄存器中的数据。它扩展了许多指令的功能，改善了内核的性能，提高了代码密度。

- Thumb 指令集

ARM 处理器根据 RICS 原理设计，但是由于各种原因，在低代码密度上它比其他多数 RICS 要好一些，然而它的代码密度仍不如某些 CISC 处理器。在代码密度重要的场合，ARM 公司在某些版本的 ARM 处理器中加入了一个称为 Thumb 结构的新型机构。Thumb 指令集是原来 32 位 ARM 指令集的 16 位压缩形式，并在指令流水线中使用了动态解压缩硬件。Thumb 代码密度优于多数 CISC 处理器达到的代码密度。

- 条件执行

只有当某个特定条件满足时指令才会被执行。这个特性可以减少分支指令数目，从而改善性能，提高代码密度。

- DSP 指令

一些功能强大的数字信号处理（DSP）指令被加入到标准的 ARM 指令中，以支持快速的 16×16 位乘法操作及饱和运算。在某些应用中，传统的方法需要微处理器加上 DSP 才能实现。这些增强指令，使得 ARM 处理器也能够满足这些应用的需要。

综上所述，ARM 体系结构的主要特征如下：

- 大量的寄存器，它们都可以用于多种用途；
- Load/Store 体系结构；
- 每条指令都条件执行；
- 多寄存器的 Load/Store 指令；
- 能够在单时钟周期执行的单条指令内完成一项普通的移位操作和一项普通的 ALU 操作；
- 通过协处理器指令集来扩展 ARM 指令集，包括在编程模式中增加了新的寄存器和数据类型。

如果把 Thumb 指令集也当作 ARM 体系结构的一部分，那么还可以加上：

- 在 Thumb 体系结构中以高密度 16 位压缩形式表示指令集。

2.2 流水线

2.2.1 流水线的概念与原理

处理器按照一系列步骤来执行每一条指令。典型的步骤如下：

① 从存储器读取指令（fetch）；

② 译码以鉴别它是属于哪一条指令（dec）；

③ 从指令中提取指令的操作数（这些操作数往往存在于寄存器中）（reg）；

④ 将操作数进行组合以得到结果或存储器地址（ALU）；

⑤ 如果需要，则访问存储器以存储数据（mem）；

⑥ 将结果写回到寄存器堆（res）。

并不是所有的指令都需要上述每一个步骤，但是，多数指令需要其中的多个步骤。这些步骤往往使用不同的硬件功能，例如，ALU 可能只在第 4 步中用到。因此，如果一条指令不是在前一条指令结束之前就开始，那么在每一步骤内处理器只有少部分的硬件在使用。

有一种方法可以明显改善硬件资源的使用率和处理器的吞吐量，这就是当前一条指令结束之前就开始执行下一条指令，即通常所说的流水线（Pipeline）技术。流水线是 RISC 处理器执行指令时采用的机制。使用流水线，可在取下一条指令的同时译码和执行其他指令，从而加快执行的速度。可以把流水线看作是汽车生产线，每个阶段只完成专门的处理器任务。

采用上述操作顺序，处理器可以这样来组织：当一条指令刚刚执行完步骤①并转向步骤②时，下一条指令就开始执行步骤①。图 2.1 说明了这个过程。从原理上说，这样的流水线应该比没有重叠的指令执行快 6 倍，但由于硬件结构本身的一些限制，实际情况会比理想状态差一些。

2.2.2 流水线的分类

从 Acorn Computer 公司在 1983～1985 年间开发的第一个 3μm 器件，到 ARM 公司在 1990～1995 年间开发的 ARM6 和 ARM7，ARM 整数处理器核的组织结构变化很小，这些处理器都是采用 3 级流水线，而这一时期 CMOS 工艺的发展，几乎将特征尺寸减少了一个数量级。因此，核的性能提高很快，但基本的操作原理大部分没有变化。

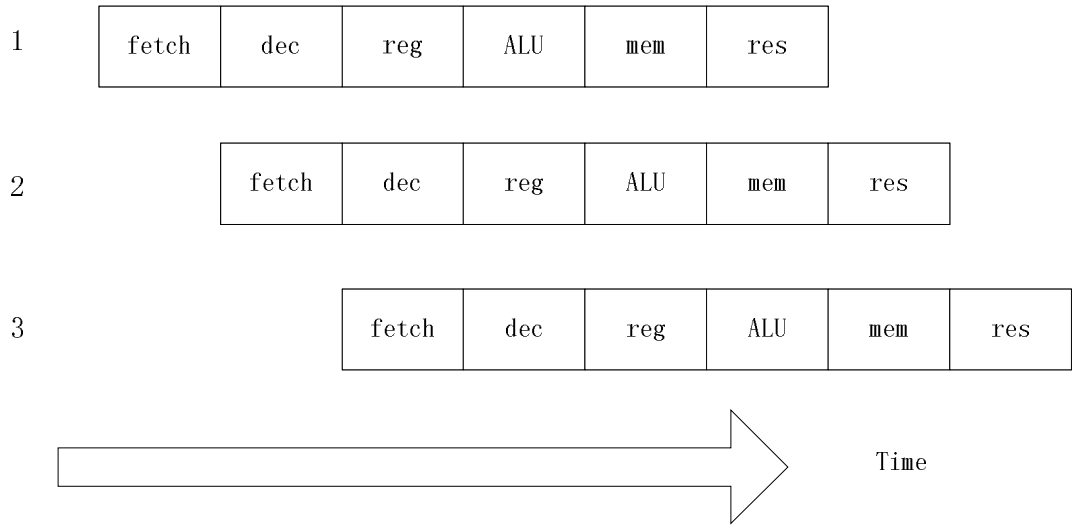


图 2.1　流水线的指令执行过程

从 1995 年以来，ARM 公司推出了几个新的 ARM 核。它们采用 5 级流水线和哈佛架构，获得了显著的高性能。例如，ARM9 增加了存储器访问段和回写段，这使得 ARM9 的处理能力可达到平均 1.1 Dhrystone[1] MISP/MHz，与 ARM7 相比，指令吞吐量提高了约 13%。

注意　在许多高性能处理器内部，一级 Cache 一般都设置有两个，其中，一个是指令 Cache，另一个是数据 Cache。这样可以减少取指令和读操作数的访问冲突，这种结构被称为哈佛架构。
把主存储器分成两个独立编址的存储器，一个专门存放指令，称为指令存储器，简称指存；另一个专门存放操作数，称为数据存储器，简称数存。两个存储器可以同时访问，这样就解决了取指令和读操作数的冲突。如果在此基础上规定在执行指令阶段产生的运算结果只写到通用寄存器中，不写到主存，那么取指令、分析指令和执行指令就可以同时进行。

ARM10 更是把流水线增加到 6 级。ARM10 的平均处理能力达到 1.3 Dhrystone MISP/MHz，与 ARM7 相比，指令吞吐量提高了约 34%。

注意　虽然 ARM9 和 ARM10 的流水线不同，但它们都使用了与 ARM7 相同的流水线执行机制，因此 ARM7 上的代码也可以在 ARM9 和 ARM10 上运行。

1．3 级流水线 ARM 组织

3 级流水线 ARM 组织如图 2.2 所示，其主要的组成如下：

① 处理器状态寄存器堆（Rigister Bank）。它有两个读端口和一个写端口，每个端口都可以访问任意寄存器。另外还有附加的可以访问 PC 的一个读端口和一个写端口。

注意　PC 的附加写端口可以在取指地址增加后更新 PC，读端口可以在数据地址发出之后从新开始取指。

[1] Dhrystone 是测量处理器运算能力的最常见基准程序之一，Dhrystone 的计量单位为每秒计算多少次 Dhrystone。

② 桶形移位寄存器（Barrel Shifter）。它可以把一个操作数移位或循环移位任意位数。
③ ALU。完成指令集要求的算术或逻辑功能。

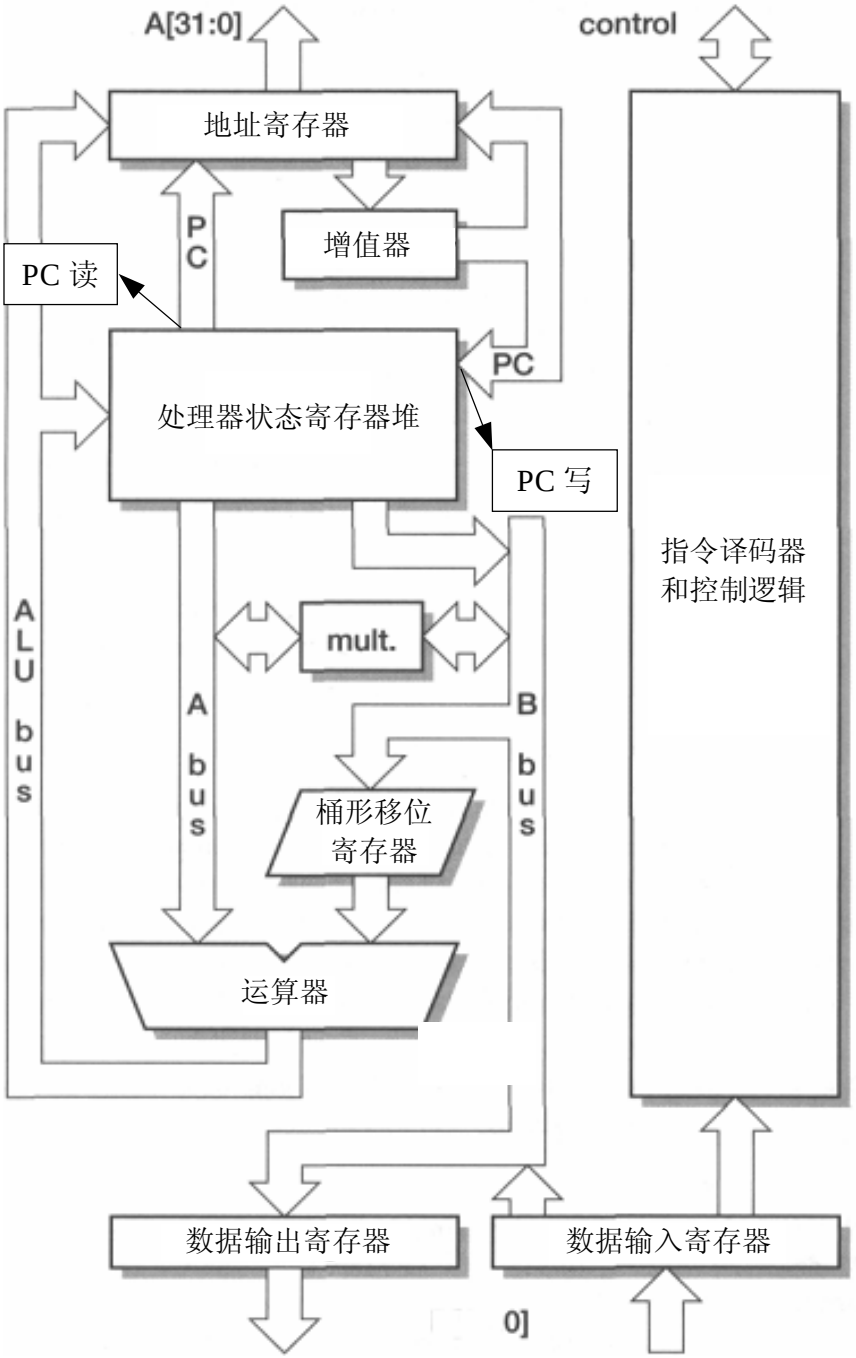


图 2.2　3 级流水线 ARM 的组织

④ 地址寄存器（Address Register）和增值器（Incrementer）。可选择和保存所用的存储

器地址并在需要时产生顺序地址。

⑤ 数据输出寄存器（data-out register）和数据输入寄存器（data-in register）。用于保存传输到存储器和从存储器输出的数据。

⑥ 指令译码器和相关的控制逻辑（instruction decode and control）。

例 2.1 显示了一条单周期指令在流水线上的执行过程。

【例 2.1】

```
ADD  r1，r2
```

指令在流水线上的执行过程如图 2.3 所示。

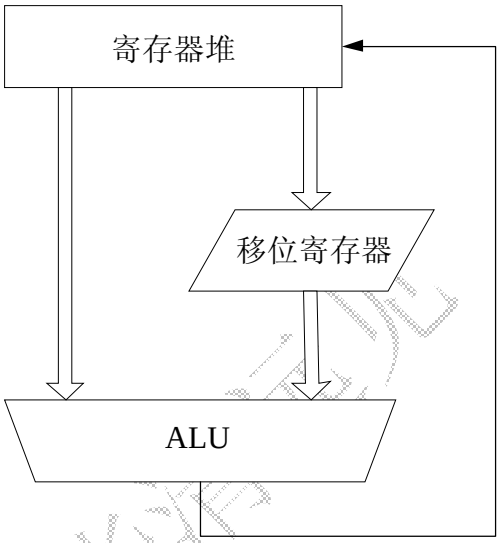


图 2.3　单周期指令在流水线上的执行过程

在 ADD 指令中，需要访问两个寄存器操作数，B 总线上的数据移位后与 A 总线上的数据在 ALU 中组合，再将结果写回寄存器堆。在指令执行过程中，程序计数器的数据放在地址寄存器中，地址寄存器的数据送入增值器。然后将增值后的数据拷贝到寄存器堆的 r15（程序计数器），同时还拷贝到地址寄存器，作为下一次取指的地址。

到 ARM7 为止的 ARM 处理器使用简单的 3 级流水线，包括下列流水线级：

- 取指（fetch）：从寄存器装载一条指令。
- 译码（decode）：识别被执行的指令，并为下一个周期准备数据通路的控制信号。在这一级，指令占有译码逻辑，不占用数据通路。
- 执行（excute）：处理指令并将结果写回寄存器。

图 2.4 显示了 3 级流水线指令执行过程。



图 2.4　3 级流水线

注意 在任一时刻，可能有 3 种不同的指令占有这 3 级中的每一级，因此，每一级中的硬件必须能够独立操作。

当处理器执行简单的数据处理指令时，流水线使得平均每个时钟周期能完成 1 条指令。但 1 条指令需要 3 个时钟周期来完成，因此，有 3 个时钟周期的延时（latency），但吞吐率（throughput）是每个周期一条指令。例 2.2 通过一个简单的例子说明了流水线的机制。

【例 2.2】

指令序列为：

```
ADD r1 r2
SUB r3 r2
CMP r1 r3
```

流水线指令序列如图 2.5 所示。

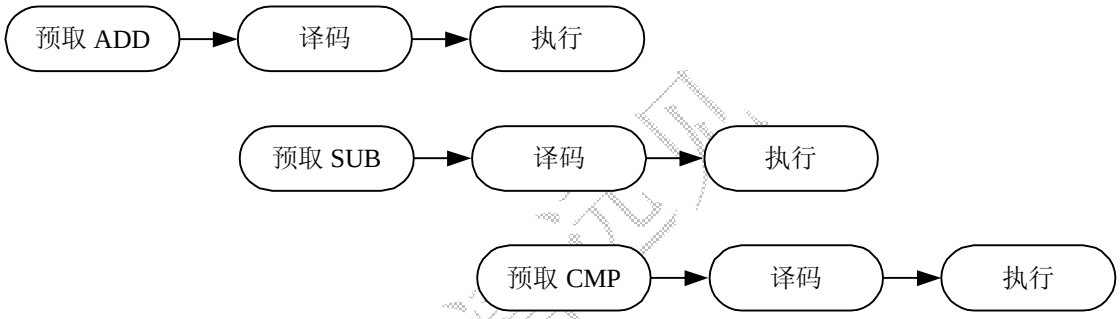


图 2.5 流水线指令顺序

在第一个周期，内核从存储器取出指令 ADD；在第二个周期，内核取出指令 SUB，同时对 ADD 译码；在第三个周期，指令 SUB 和 ADD 都沿流水线移动，ADD 被执行，而 SUB 被译码，同时又取出 CMP 指令。可以看出，流水线使得每个时钟周期都可以执行一条指令。

当执行多条指令时，流水线的执行不一定会如图 2.5 那么规则，图 2.6 显示了有 STR 指令的流水线状态。

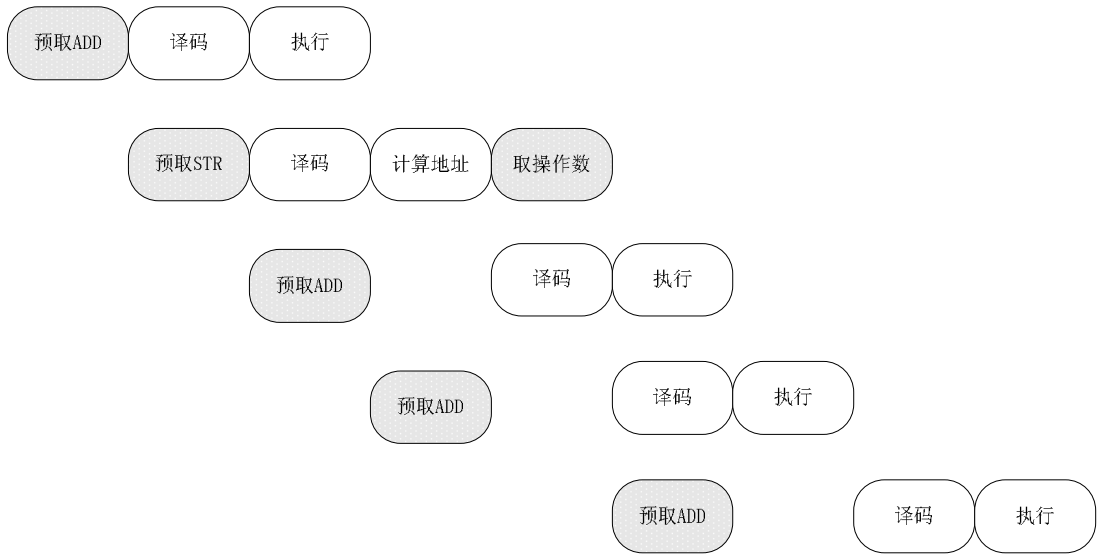


图 2.6　含有存储器访问指令的流水线状态

图 2.6 中在单周期指令 ADD 后出现了一条数据存储指令 STR。访问主存储器的指令用阴影表示，可以看出在每个周期都使用了存储器。同样，在每一个周期也使用了数据通路。在执行周期、地址计算和数据传输周期，数据通路都是被占用的。在译码周期，译码逻辑负责产生下一周期用到的数据通路的控制信号。

注意　对于 STR 这种存储器访问指令，实际是在地址计算时由译码逻辑产生下一周期数据传输所需要的数据通路控制信号。

在图 2.6 中的指令序列中，处理器的每个逻辑单元在每个指令都是活动的。可以看出流水线的执行与存储器访问密切相关。存储器访问限制了程序执行必须花费的指令周期数。

ARM 的流水线执行模式导致了一个结果，就是程序计数器 PC（对使用者而言为 r15）必须在当前指令执行前计数。例如，指令在其第一个周期为下下条指令取指，这就意味着 PC 必须指向当前指令的后 8 个字节（其后的第 2 条指令）。

当程序中必须用到 PC 时，程序员要特别注意这一点。大多数正常情况下，不用考虑这一点，它由汇编器或编译器自动处理这些细节。

例 2.3 显示了流水线下程序计数器 PC 的使用情况。

【例 2.3】

指令序列为：

```
0x8000 LDR pc，[pc，＃0]
0x8004 NOP
0x8008 DCD jumpAdress
```

当指令 LDR 处于执行阶段时，pc=address+8 即 0x8008。

2．5 级流水线 ARM 组织

所有的处理器都要满足对高性能的要求。直到 ARM7 为止，在 ARM 核中使用的 3 级流水线的性价比是很高的。但是，为了得到更高的性能，需要重新考虑处理器的组织结构。执行一个给定的程序需要的时间由下式决定：

$$T_{prog} = (N_{inst} \times CPI) / f_{clk}$$

式中：

N_{inst}：表示在程序中执行的 ARM 指令数；

CPI：表示每条指令的平均时钟周期；

f_{clk}：表示处理器的时钟频率。

因为对给定程序（假设使用给定的优化集并用给定的编译器来编译）N_{inst} 是常数，所以，仅有两种方法来提供性能。

第一，提高时钟频率。时钟频率的提高，必然引起指令执行周期的缩短，所以要求简化流水线每一级的逻辑，流水线的级数就要增加。

第二，减少每条指令的平均指令周期数 CPI。这就要求重新考虑 3 级流水线 ARM 中多于 1 个流水线周期的实现方法，以便使其占有较少的周期，或者减少因指令相关造成的流水线停顿，也可以将两者结合起来。

3 级流水线 ARM 核在每一个时钟周期都访问存储器，或者取指令，或者传输数据。只是抓紧存储器不用的几个周期来改善系统系统性能，效果是不明显的。为了改善 CPI，存储器系统必须在每个时钟周期中给出多于一个的数据。方法是在每个时钟周期从单个存储器中给出多于 32 位数据，或者为指令或数据分别设置存储器。

基于以上原因，较高性能的 ARM 核使用了 5 级流水线，而且具有分开的指令和数据存储器。把指令的执行分割为 5 部分而不是 3 部分，进而可以使用更高的时钟频率，分开的指令和数据存储器使核的 CPI 明显减少。

注意 分开的指令和数据存储器。一般是分开的 Cache 连接到统一的指令和数据存储器上。

在 ARM9TDMI 中使用了典型的 5 级流水线。ARM9TDMI 的组织结构如图 2.7 所示。

5 级流水线包括下面的流水线级：

- 取指（fetch）：从存储器中取出指令，并将其放入指令流水线。
- 译码（decode）：指令被译码，从寄存器堆中读取寄存器操作数。在寄存器堆中有 3 个操作数读端口，因此，大多数 ARM 指令能在 1 个周期内读取其操作数。
- 执行（execute）：将其中一个操作数移位，并在 ALU 中产生结果。如果指令是 Load 或 Store 指令，则在 ALU 中计算存储器的地址。
- 缓冲/数据（buffer/data）：如果需要则访问数据存储器，否则 ALU 只是简单地缓冲一个时钟周期。
- 回写（write-back）：将指令的结果回写到寄存器堆，包括任何从寄存器读出的数据。

图 2.8 显示了 5 级流水线指令的执行过程。

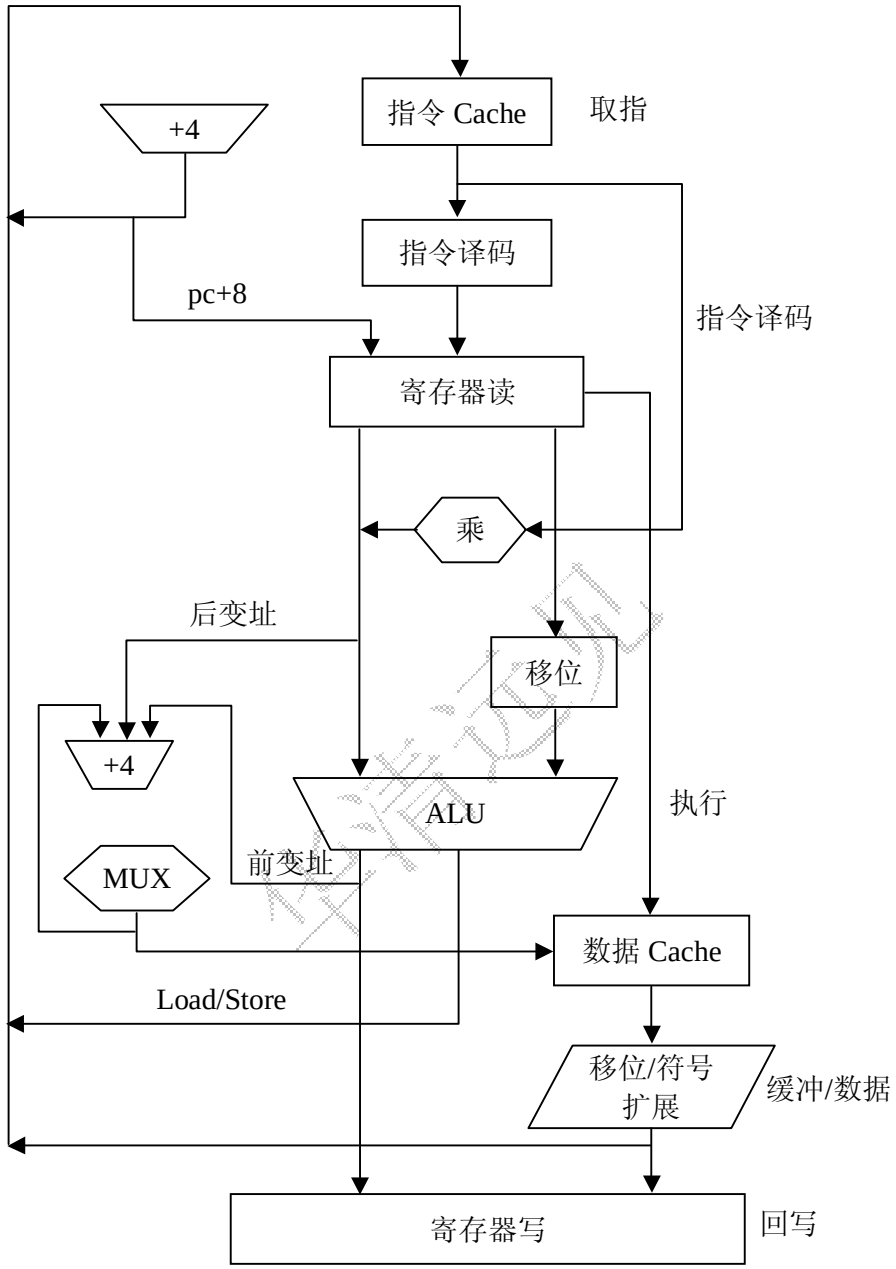


图 2.7　5 级流水线的组织结构



图 2.8　5 级流水线

在程序执行过程中，PC 值是基于 3 级流水线操作特性的。5 级流水线中提前 1 级来读取指令操作数，得到的值是不同的（PC+4 而不是 PC+8）。这产生的代码不兼容是不容许的。但 5 级流水线 ARM 完全仿真 3 级流水线的行为。在取指级增加的 PC 值被直接送到译码级的寄存器，穿过两极之间的流水线寄存器。下一条指令的 PC+4 等于当前指令的 PC+8，因此，未使用额外的硬件便得到了正确的 r15。

3．6 级流水线 ARM 组织

在 ARM10 中，将流水线的级数增加到 6 级，使系统的平均处理能力达到了 1.3Dhrystone MISP/MHz。图 2.9 显示了 6 级流水线上指令的执行过程。

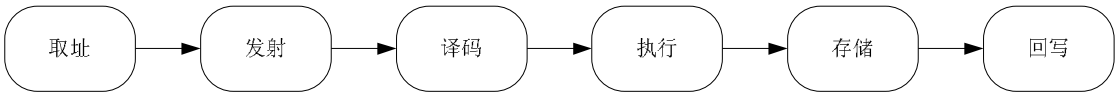


图 2.9　6 级流水线

2.2.3　影响流水线性能的因素

1．互锁

在典型的程序处理过程中，经常会遇到这样的情形，即一条指令的结果被用做下一条指令的操作数。如例 2.4 所示。

【例 2.4】

有如下指令序列：

```
LDR r0,[r0,#0]
ADD r0,r0,r1     ;在 5 级流水线上产生互锁
```

从例 2.4 中可以看出，流水线的操作产生中断，因为第一条指令的结果在第二条指令取数时还没有产生。第二条指令必须停止，直到结果产生为止。

2．跳转指令

跳转指令也会破坏流水线的行为，因为后续指令的取指步骤受到跳转目标计算的影响，因而必须推迟。但是，当跳转指令被译码时，在它被确认是跳转指令之前，后续的取指操作已经发生。这样一来，已经被预取进入流水线的指令不得不被丢弃。如果跳转目标的计算是在 ALU 阶段完成的，那么，在得到跳转目标之前已经有两条指令按原有指令流读取。

解决的办法是，如果有可能最好早一些计算转移目标，当然这需要硬件支持；如果转移指令具有固定格式，那么可以在解码阶段预测跳转目标，从而将跳转的执行时间减少到单个周期。但要注意，由于条件跳转与前一条指令的条件码结果有关，在这个流水线中，还会有条件转移的危险。

尽管有些技术可以减少这些流水线问题的影响，但是，不能完全消除这些困难。流水线

级数越多，问题就越严重。对于相对简单的处理器，使用 3～5 级流水线效果最好。

显然，只有当所有指令都依照相似的步骤执行时，流水线的效率达到最高。如果处理器的指令非常复杂，每一条指令的行为都与下一条指令不同，那么就很难用流水线实现。

2.3 ARM 存储器

ARM 处理器内核广泛应用于嵌入式系统和其他行业应用中。为了适应不同系统的需要，ARM 采用了灵活多样的存储管理体系。从平板式内存映射到灵活方便的 MMU 内存管理单元，用户可以根据自己的需要使用不同的存储管理策略。

在 ARM 体系结构中可使用的存储管理策略包括：

- 多类型的存储单元（可以使用 SDRAM、FLASH 等）；
- Cache；
- 写缓存；
- 虚拟内存地址。

另外，内存映射 I/O 机制可以使开发者灵活、方便地增加大量外设。

可以通过下面的几种方法实现对存储系统的管理：

- 使能 Cache，加快存储器的访问速度；
- 启动虚拟地址到物理地址的映射；
- 使用“域管理”策略，对存储单元的访问进行保护；
- 对 I/O 映射地址空间的访问加以限制。

标准的对 ARM 处理器的存储管理是使用协处理器 CP15 来实现的。ARM 体系的存储系统将在第 15 章详细介绍。

2.4 I/O 管理

ARM 系统完成 I/O 功能的标准方法是使用存储器映射 I/O。这种方法使用特定的存储器地址。当从这些地址加载或向这些地址存储时，它们提供 I/O 功能。某些 ARM 系统也可能有直接存储器访问（DMA，Direct Memory Access）硬件。

外围设备（如串行线控制器）中包含一些寄存器。在存储器映射系统中，这些寄存器就像特定地址的存储器一样。（在其他的系统组织中，I/O 功能可能与存储器件有不同的寻址空间。）串行线控制器可能有以下 5 种寄存器。

① 发送数据寄存器（只写）：写入这个位置的数据被送往串行线。

② 接受数据寄存器（只读）：保存从串行线送来的数据。

③ 控制寄存器（读/写）：设置数据速率，管理 RTS（请求发送）和其他类似信号。

④ 中断使能寄存器（读/写）：控制中断的硬件事件。

⑤ 状态寄存器（读/写）：指示读数据是否有效、写缓存是否满等。

要接受数据，必须用软件适当地设置器件。通常在接收到有效数据或检测到错误时产生一个中断。中断程序必须将数据复制到缓存器中并进行错误检测。

应该注意的是，存储器映射外围寄存器的行为与存储器不同。连续两次读数据寄存器，即使对该寄存器没有写操作，其结果也很可能不同。而对真正存储器的读是幂等的（idempotent）（可多次重复读，结果一致）。对外围寄存器的读操作可能清除当前值，致使下一次读结果不同。这种寄存器称为读敏感（read-sensitive）的。

当涉及读敏感寄存器时，编程必须小心。特别是不能将这种寄存器的数据复制到 Cache 存储器。

在许多 ARM 系统中，不能在用户模式下访问 I/O 寄存器。要访问这些器件，只能通过监控调用（SWI）或通过使用这种调用的 C 库函数。

注意 在 ARM 编程中，通常将存储器的 I/O 区域标记为非 Cache 区（uncacheable），并绕过 Cache 访问。通常 Cache 与读敏感（read－sensitive）器件相互排斥。显示帧缓存器（DisplayFrame Buffers）也需要仔细考虑，通常也设为不可 Cache 的。

2.5 ARM 开发调试方法

用户选用 ARM 处理器开发嵌入式系统时，选择合适的开发工具可以加快开发进度，节省开发成本。因此一套含有编辑软件、编译软件、汇编软件、链接软件、调试软件、工程管理及函数库的集成开发环境（IDE）一般来说是必不可少的，如 ARM 公司的 RealView 开发环境。至于嵌入式实时操作系统、评估板等其他开发工具则可以根据应用软件规模和开发计划选用。

使用集成开发环境开发基于 ARM 的应用软件，包括编辑、编译、汇编、链接等工作全部在 PC 机上即可完成，调试工作则需要配合其他的模块或产品方可完成。目前常见的调试方法有以下几种。

1．指令集模拟器

部分集成开发环境提供了指令集模拟器，可方便用户在 PC 机上完成一部分简单的调试工作。但是，由于指令集模拟器与真实的硬件环境相差很大，因此即使用户使用指令集模拟器调试通过的程序也有可能无法在真实的硬件环境下运行，用户最终必须在硬件平台上完成整个应用的开发。

2．驻留监控软件

驻留监控软件（Resident Monitors）是一段运行在目标板上的程序，集成开发环境中的调试软件通过以太网口、并行端口、串行端口等通信端口与驻留监控软件进行交互，由调试软件发布命令通知驻留监控软件控制程序的执行、读写存储器、读写寄存器、设置断点等。

利用驻留监控软件是一种比较低廉有效的调试方式，不需要任何其他的硬件调试和仿真设备。ARM 公司的 Angel 就是该类软件，大部分嵌入式实时操作系统也采用该类软件进行调试，不同的是在嵌入式实时操作系统中，驻留监控软件是作为操作系统的一个任务存在的。

驻留监控软件的不便之处在于它对硬件设备的要求比较高，一般在硬件稳定之后才能进

行应用软件的开发，同时它占用目标板上的一部分资源，而且不能对程序的全速运行进行完全仿真，所以对一些要求严格的情况不是很适合。

3．JTAG 仿真器

JTAG 仿真器也称为 JTAG 调试器，是通过 ARM 芯片的 JTAG 边界扫描口进行调试的设备。JTAG 仿真器比较便宜，连接比较方便，通过现有的 JTAG 边界扫描口与 ARM 处理器核通信，属于完全非插入式（即不使用片上资源）调试。它无需目标存储器，不占用目标系统的任何端口，而这些是驻留监控软件所必需的。另外，由于 JTAG 调试的目标程序是在目标板上执行，仿真更接近于目标硬件，因此，许多接口问题，如高频操作限制、AC 和 DC 参数不匹配、电缆长度的限制等被最小化了。使用集成开发环境配合 JTAG 仿真器进行开发是目前采用最多的一种调试方式。

4．在线仿真器

在线仿真器使用仿真头完全取代目标板上的 ARM 处理器，可以完全仿真 ARM 芯片的行为，提供更进一步的调试功能。但这类仿真器为了能够全速仿真时钟速度高于 100MHz 的处理器，通常必须采用极其复杂的设计和工艺，因而其价格比较昂贵。在线仿真器通常用在 ARM 的硬件开发中，在软件的开发中较少使用。其价格高昂是在线仿真器难以普及的原因。

华清远见

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 3 章 ARM 微处理器的编程模型

华清远见

3.1 数据类型

3.1.1 ARM 的基本数据类型

ARM 采用的是 32 位架构，ARM 的基本数据类型有以下 3 种。

- Byte：字节，8bit。
- Halfword：半字，16bit（半字必须于 2 字节边界对齐）。
- Word：字，32bit（字必须于 4 字节边界对齐）。

存储器可以看作是序号为 0～2^{32}–1 的线性字节阵列。图 3.1 所示为 ARM 存储器的组织结构。

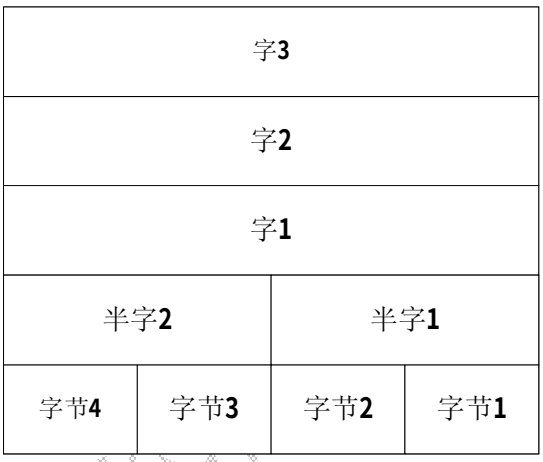


图 3.1 ARM 存储器组织结构

图 3.1 所示为存储器的一小片区域，其中每一个字节都有惟一的地址。字节可以占用任一位置，图中给出了几个例子。长度为 1 个字的数据项占用一组 4 字节的位置，该位置开始于 4 的倍数的字节地址（地址最末两位为 00）。图 3.1 中包含了 3 个这样的例子。半字占有两个字节的位置，该位置开始于偶数字节地址（地址最末一位为 0）。

注意

① ARM 系统结构 v4 以上版本支持以上 3 种数据类型，v4 以前版本仅支持字节和字。

② 当将这些数据类型中的任意一种声明成 unsigned 类型时，*N* 位数据值表示范围为 0～2^n–1 的非负数，通常使用二进制格式。

③ 当将这些数据类型的任意一种声明成 signed 类型时，*N* 位数据值表示范围为–2^{n-1}～2^{n-1}–1 的整数，使用二进制的补码格式。

④ 所有数据类型指令的操作数都是字类型的，如“ADD r1，r0，＃0x1”中的操作数“0x1”就是以字类型数据处理的。

⑤ Load/Store 数据传输指令可以从存储器存取传输数据，这些数据可以是字节、半字、字。加载时自动进行字节或半字的零扩展或符号扩展。对应的指令分别为 LDR/BSTRB（字节操作）、LDRH/STRH（半字操作）、LDR/STR（字操作）。详见后面的指令参考。

⑥ ARM 指令编译后是 4 个字节（与字边界对齐）。Thumb 指令编译后是 2 个字节（与半字边界对齐）。

3.1.2 浮点数据类型

浮点运算使用在 ARM 硬件指令集中未定义的数据类型。

尽管如此，但 ARM 公司在协处理器指令空间定义了一系列浮点指令。通常这些指令全部可以通过未定义指令异常（此异常收集所有硬件协处理器不接受的协处理器指令）在软件中实现，但是其中的一小部分也可以由浮点运算协处理器 FPA10 以硬件方式实现。

另外，ARM 公司还提供了用 C 语言编写的浮点库作为 ARM 浮点指令集的替代方法（Thumb 代码只能使用浮点指令集）。该库支持 IEEE 标准的单精度和双精度格式。C 编译器有一个关键字标志来选择这个历程。它产生的代码与软件仿真（通过避免中断、译码和浮点指令仿真）相比既快又紧凑。

3.1.3 存储器大/小端

从软件角度看，内存相对于一个大的字节数组，其中每个数组元素（字节）都是可寻址的。

ARM 支持大端模式（big-endian）和小端模式（little-endian）两种内存模式。

图 3.2 和图 3.3 分别显示了内存的大端模式和小端模式。

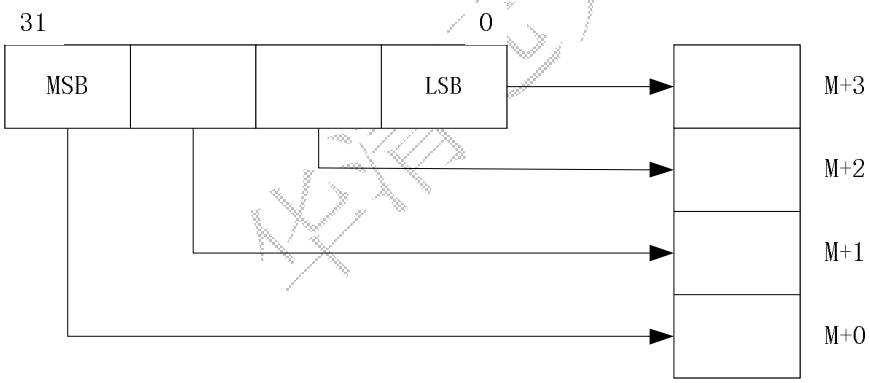


图 3.2　大端模式

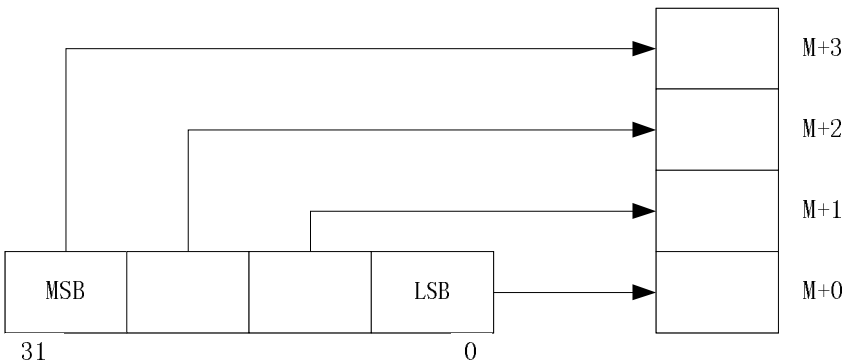


图 3.3　小端模式

下面的例子显示了使用内存大/小端（big/little endian）的存取格式。

【例 3.1】

程序执行前：

```
r0=0x11223344
```

执行指令：

```
r1=0x100
STR r0，[r1]
LDRB r2，[r1]
```

执行后：

```
小端模式下：r2=0x44
大端模式下：r2=0x11
```

上面的例子向我们提示了一个潜在的编程隐患。在大端模式下，一个字的高地址放的是数据的低位，而在小端模式下，数据的低位放在内存中的低地址。要小心对待存储器中一个字内字节的顺序。

3.2 处理器工作模式

ARM 处理器共有 7 种工作模式，如表 3.1 所示。

表 3.1 ARM 处理器的工作模式

处理器工作模式	简 写	描 述
用户模式（User）	usr	正常程序执行模式，大部分任务执行在这种模式下
快速中断模式（FIQ）	fiq	当一个高优先级（fast）中断产生时将会进入这种模式，一般用于高速数据传输和通道处理
外部中断模式（IRQ）	irq	当一个低优先级（normal）中断产生时将会进入这种模式，一般用于通常的中断处理
特权模式（Supervisor）	svc	当复位或软中断指令执行时进入这种模式，是一种供操作系统使用的保护模式
数据访问中止模式（Abort）	abt	当存取异常时将会进入这种模式，用于虚拟存储或存储保护
未定义指令中止模式（Undef）	und	当执行未定义指令时进入这种模式，有时用于通过软件仿真协处理器硬件的工作方式
系统模式（System）	sys	使用和 User 模式相同寄存器集的模式，用于运行特权级操作系统任务

除用户模式外的其他 6 种处理器模式称为特权模式（Privileged Modes）。在这些模式下，程序可以访问所有的系统资源，也可以任意地进行处理器模式切换。其中的 5 种又称为异常模式，分别为：

- FIQ(Fast Interrupt reQuest)；
- IRQ(Interrupt request)；
- 管理（Supervisor)；
- 中止（Abort)；
- 未定义（Undefined)。

处理器模式可以通过软件控制进行切换，也可以通过外部中断或异常处理过程进行切换。

大多数的用户程序运行在用户模式下。当处理器工作在用户模式时，应用程序不能够访问受操作系统保护的一些系统资源，应用程序也不能直接进行处理器模式切换。当需要进行处理器模式切换时，应用程序可以产生异常处理，在异常处理过程中进行处理器模式切换。这种体系结构可以使操作系统控制整个系统资源的使用。

当应用程序发生异常中断时，处理器进入相应的异常模式。在每一种异常模式中都有一组专用寄存器以供相应的异常处理程序使用，这样就可以保证在进入异常模式时用户模式下的寄存器（保存程序运行状态）不被破坏。

系统模式，不能有任何异常进入。仅 ARM 体系结构 v4 及以上版本有该模式。它和用户模式具有完全相同的寄存器。但是系统模式属于特权模式，可以访问所有的系统资源，也可以直接进行处理器模式切换，它主要供操作系统任务使用。通常操作系统的任务需要访问所有的系统资源，同时该任务仍然使用用户模式的寄存器组而不是异常模式下相应的寄存器组，这样可以保证当异常中断发生时任务状态不被破坏。

3.3 ARM 寄存器组织

ARM 处理器有 37 个 32 位长的寄存器。

- 1 个用作 PC（Program Counter)。
- 1 个用作 CPSR(Current Program Status Register)。
- 5 个用作 SPSR（Saved Program Status Registers)。
- 30 个用作通用寄存器。

注意 以上 37 个寄存器中，1 个 CPSR 和 5 个 SPSR 通称为状态寄存器，虽然这些寄存器是 32 位的，但目前只使用了其中的 12 位。除了这 6 个状态寄存器外，其余的 31 个寄存器又称为通用寄存器。

ARM 处理器共有 7 种不同的处理器模式，在每一种处理器模式中有一组相应的寄存器组。表 3.2 显示了 ARM 的寄存器组织概要。

表 3.2 寄存器组织概要

User	FIQ	IRQ	SVC	Undef	Abort
R0	User mode R0～R7,R15,and CPSR	User mode R0～R12,R15 and CPSR	User mode R0～R12,R15 and CPSR	User mode R0～R12,R15 and CPSR	User mode R0～R12,R15 and CPSR
R1					
R2					
R3					

R4										
R5										
R6										
R7										
R8		R8								
R9		R9								
R10		R10								

续表

User		FIQ		IRQ		SVC		Undef		Abort
R11		R11								
R12		R12								
R13(SP)		R13(SP)		R13		R13		R13		R13
R14(LR)		R14(LR)		R14		R14		R14		R14
R15(PC)										
CPSR										
		SPSR		SPSR		SPSR		SPSR		SPSR

注意 System 模式使用和 User 模式相同的寄存器集

当前处理器的模式决定着哪组寄存器可操作，任何模式都可以存取。

- 相应的 r0～r12。
- 相应的 r13（the stack pointer, sp）和 r14（the link register, lr）。
- 相应的 r15（the program counter, pc）。
- 相应的 CPSR（current program status register, cpsr）。

特权模式（除 System 模式）还可以存取。

- 相应的 SPSR（saved program status register）。

3.3.1 通用寄存器

通用寄存器根据其分组与否和使用目的分为以下 3 类。

- 未分组寄存器（The unbanked registers），包括 r0～r7。
- 分组寄存器（The banked register），包括 r8～r14。
- 程序计数器（Program Counter），即 r15。

1．未分组寄存器

未分组寄存器包括 r0～r7。顾名思义，在所有处理器模式下对于每一个未分组寄存器来说，指的都是同一个物理寄存器。未分组寄存器没有被系统用于特殊的用途，任何可采用通用寄存器的应用场合都可以使用未分组寄存器。但由于其通用性，在异常中断所引起的处理

器模式切换时，其使用的是相同的物理寄存器，所以也就很容易使寄存器中的数据被破坏。

2. 分组寄存器

r8～r14 是分组寄存器，它们每一个访问的物理寄存器取决于当前的处理器模式。

对于这些分组寄存器 r8～r12 来说，每个寄存器对应两个不同的物理寄存器。一组用于除 FIQ 模式外的所有处理器模式，而另一组则专门用于 FIQ 模式。这样的结构设计有利于加快 FIQ 的处理速度。不同模式下寄存器的使用，要使用寄存器名后缀加以区分，例如，当使用 FIQ 模式下的寄存器时，寄存器 r8 和寄存器 r9 分别记做 r8_fiq, r9_fiq；当使用用户模式下的寄存器时，寄存器 r8 和 r9 分别记做 r8_usr, r9_usr 等。在 ARM 体系结构中，r8～r12 没有任何指定的其他的用途，所以当 FIQ 中断到达时，不用保存这些通用寄存器，也就是说 FIQ 处理程序可以不必执行保存和恢复中断现场的指令，从而可以使中断处理过程非常迅速。所以 FIQ 模式常被用来处理一些时间紧急的任务，如 DMA 处理。

对于分组寄存器 r13 和 r14 来说，每个寄存器对应 6 个不同的物理寄存器。其中的一个是用户模式和系统模式公用的，而另外 5 个分别用于 5 种异常模式。访问时需要指定它们的模式。名字形式如下：

- r13_<mode>
- r14_<mode>

其中<mode>可以是以下几种模式之一：usr、svc、abt、und、irp 及 fiq。

r13 寄存器在 ARM 中常用作堆栈指针，称为 SP。当然，这只是一种习惯用法，并没有任何指令强制性的使用 r13 作为堆栈指针，用户完全可以使用其他寄存器作为堆栈指针。而在 Thumb 指令集中，有一些指令强制性的将 r13 作为堆栈指针，如堆栈操作指令。

每一种异常模式拥有自己的 r13。异常处理程序负责初始化自己的 r13，使其指向该异常模式专用的栈地址。在异常处理程序入口处，将用到的其他寄存器的值保存在堆栈中，返回时，重新将这些值加载到寄存器。通过这种保护程序现场的方法，异常不会破坏被其中断的程序现场。

寄存器 r14 又被称为连接寄存器（Link Register，LR），在 ARM 体系结构中具有下面两种特殊的作用。

（1）每一种处理器模式用自己的 r14 存放当前子程序的返回地址。当通过 BL 或 BLX 指令调用子程序时，r14 被设置成该子程序的返回地址。在子程序返回时，把 r14 的值复制到程序计数器 PC。典型的做法是使用下列两种方法之一。

- 执行下面任何一条指令。

```
MOV  PC, LR
BX  LR
```

- 在子程序入口处使用下面的指令将 PC 保存到堆栈中。

```
STMFD  SP!, {<register>,LR}
```

在子程序返回时，使用如下相应的配套指令返回。

```
LDMFD  SP!, {<register>,PC}
```

（2）当异常中断发生时，该异常模式特定的物理寄存器 r14 被设置成该异常模式的返回

地址，对于有些模式 r14 的值可能与返回地址有一个常数的偏移量（如数据异常使用 SUB PC, LR,#8 返回）。具体的返回方式与上面的子程序返回方式基本相同，但使用的指令稍微有些不同，以保证当异常出现时正在执行的程序的状态被完整保存。

R14 也可以被用作通用寄存器使用。

注意

当嵌套中断被允许时（即异常可重入），r13 和 r14 的使用要特别小心。例如，在用户模式下一个 IRQ 中断发生，这时两种模式分别使用不同的 r13 和 r14，换句话讲，用户模式使用 r13_usr 和 r14_usr，而 IRQ 模式使用 r13_irq 和 r14_irq，这样不会造成寄存器使用冲突。但是，当程序运行在 IRQ 模式下，又有 IRQ 中断进入，此时，第二级中断使用 r13_irq 和 r14_irq，冲掉了第一级 IRQ 的堆栈指针和返回地址，导致程序异常。

解决办法是在第二级中断发生前，将第一级中断用到的寄存器压栈。

3.3.2 程序计数器 r15

程序计算器 r15 又被记为 PC。它有时可以被和 r0－r14 一样用作通用寄存器，但很多特殊的指令在使用 r15 时有些限制。当违反了这些指令的使用限制时，指令的执行结果是不可预知的。

程序计数器在下面两种情况下用于特殊的目的。

- 读程序计数器。
- 写程序计数器。

1．程序计数器读操作

由于 ARM 的流水线机制，指令读出的 r15 的值是指令地址加上 8 个字节。由于 ARM 指令始终是字对齐的，所以读出的结果位[1：0]始终是 0（但在 Thumb 状态下，指令为 2 字节对齐，bit[0]=0）。

读 PC 主要用于快速地对临近的指令或数据进行位置无关寻址，包括程序中的位置无关分支。

需要注意的是，当使用指令 STR 或 STM 对 r15 进行保存时，保存的可能是当前指令地址加 8 或当前指令地址加 12。到底是哪种方式，取决于芯片的具体设计方式。当然，在同一个芯片中，要么采用当前指令地址加 8，要么采用当前指令地址加 12，不可能有些指令采用当前地址加 8，有些采用当前地址加 12。程序开发人员应尽量避免使用 STR 或 STM 指令来对 r15 进行操作。当不可避免要使用这种方式时，可以先通过一小段程序来确定所使用的芯片是使用哪种方式实现的。例如：

```
SUB  R1,PC,#4          ;r1 中存放 STR 指令地址
STR  PC,[R0]           ;将 PC=STR 地址+offset 保存到 r0 中
LDR  R0,[R0]
SUB  R0,R0,R1          ;offset=PC－STR 地址
```

2．程序计数器写操作

当指令向 r15 写入地址数据时，如果指令成功返回，它将使程序跳转到该地址执行。由于 ARM 指令是字对齐的，写入 r15 的地址值应满足 bit[1：0]=0b00，具体的规则根据 ARM 版本的不同也有所不同：

- 对于 ARM 版本 3 以及更低的版本，写入 r15 的地址值 bit[1：0]被忽略，即写入 r15 的地址值将与 0xFFFFFFFC 做与操作。
- 对于 ARM 版本 4 以及更高的版本，程序必须保证写入 r15 寄存器的地址值的 bit[1：0]为 0b00，否则将会产生不可预知的结果。

对于 Thumb 指令集来说，指令是半字对齐的。处理器将忽略 bit[0]，即写入 r15 寄存器的值在写入前要先和 0XFFFFFFFE 做与操作。

有些指令对 r15 的操作有特殊的要求。比如，指令 BX 利用 bit[0]来确定需要跳转到的子程序是 ARM 状态还是 Thumb 状态。

注意 这种读取 PC 值和写入 PC 值的不对称操作需要特别注意。

3.3.3 程序状态寄存器

当前程序状态寄存器 CPSR（Current Program Status Register）可以在任何处理器模式下被访问，它包含下列内容。

- ALU（Arithmetic Logic Unit）状态标志的备份。
- 当前的处理器模式。
- 中断使能标志。
- 设置处理器的状态（只在 4T 架构）。

每一种处理器模式下都有一个专用的物理寄存器作备份程序状态寄存器 SPSR（Saved Program Status Register）。当特定的异常中断发生时，这个物理寄存器负责存放当前程序状态寄存器的内容。当异常处理程序返回时，再将其内容恢复到当前程序状态寄存器。

注意 由于用户模式和系统模式不属于异常中断模式，所以它们没有 SPSR。当在用户模式或系统模式中访问 SPSR，将会产生不可预知的结果。

CPSR 寄存器（和保存它的 SPSR 寄存器）中的位分配如图 3.4 所示。

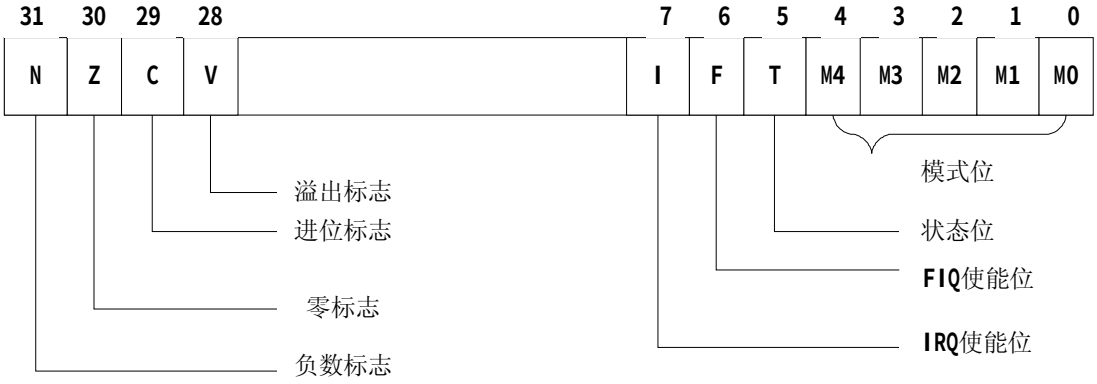


图 3.4　程序状态寄存器格式

1．标志位

N（Negative）、Z（Zero）、C（Carry）和 V（oVerflow）通称为条件标志位。这些条件标志位会根据程序中的算术或逻辑指令的执行结果进行修改，而且这些条件标志位可由大多数指令检测以决定指令是否执行。

在 ARM 4T 架构中，所有的 ARM 指令都可以条件执行，而 Thumb 指令却不能。

各条件标志位的具体含义如下。

- N

本位设置成当前指令运行结果的 bit[31]的值。当两个由补码表示的有符号整数运算时，N=1 表示运算的结果为负数；N=0 表示结果为正数或零。

- Z

Z=1 表示运算的结果为零，Z=0 表示运算的结果不为零。

注意　对于 CMP 指令，Z=1 表示进行比较的两个数相等。

- C

下面分 4 种情况讨论 C 的设置方法。

① 在加法指令中（包括比较指令 CMN），当结果产生了进位，则 C=1，表示无符号数运算发生上溢出；其他情况下 C=0。

② 在减法指令中（包括比较指令 CMP），当运算中发生错位（即无符号数运算发生下溢出），则 C=0,；其他情况下 C=1。

③ 对于在操作数中包含移位操作的运算指令（非加/减法指令），C 被设置成被移位寄存器最后移出去的位。

④ 对于其他非加/减法运算指令，C 的值通常不受影响。

- V

下面分两种情况讨论 V 的设置方法。

① 对于加/减运算指令，当操作数和运算结果都是以二进制的补码表示的带符号的数时，V=1 表示符号位溢出。

② 对于非加/减法指令，通常不改变标志位 V 的值（具体可参照 ARM 指令手册）。

尽管以上 C 和 V 的定义看起来颇为复杂，但使用时在大多数情况下用一个简单的条件测试指令即可，不需要程序员计算出条件码的精确值即可得到需要的结果。

注意

下面两种情况会对 CPSR 的条件标志位产生影响。

1. 比较指令（CMN、CMP、TEQ、TST）。
2. 目的寄存器不是 r15 的算术逻辑运算和数据传输指令。这些指令可以通过在指令末尾加标志“S”来通知处理器指令的执行结果影响标志位。

【例 3.2】

使用 SUBS 指令从寄存器 r1 中减去常量 1，然后把结果写回到 r1，其中 CPSR 的 Z 位将受到影响。

指令执行前：

```
CPSR 中 Z=0
r1=0x00000001


SUBS  r1，r1，＃1
```

SUB 指令执行结束后：

```
r1=0x0
CPSR 中 Z=1
```

目的寄存器是 r15 的带“位设置”的算术和逻辑运算指令，也可以将 SPSR 的值复制到 CPSR 中，这种操作主要用于从异常中断程序中返回。

用 MSR 指令向 CPSR/SPSR 写进新值。

目的寄存器位 r15 的 MRC 协处理器指令通过这条指令可以将协处理器产生的条件标志位的值传送到 ARM 处理器。

在中断返回时，使用 LDR 指令的变种指令可以将 SPSR 的值复制到 CPSR 中。

2．Q 标志位

在带 DSP 指令扩展的 ARM v5 及更高版本中，bit[27]被指定用于指示增强的 DAP 指令是否发生了溢出，因此也就被称为 Q 标志位。同样，在 SPSR 中 bit[27]也被称为 Q 标志位，用于在异常中断发生时保存和恢复 CPSR 中的 Q 标志位。

在 ARM v5 以前的版本及 ARM v5 的非 E 系列处理器中，Q 标志位没有被定义。属于待扩展的位。

3．控制位

CPSR 的低 8 位（I、F、T 及 M[4：0]）统称为控制位。当异常发生时，这些位的值将发生相应的变化。另外，如果在特权模式下，也可以通过软件编程来修改这些位的值。

① 中断禁止位

I=1，IRQ 被禁止。

F=1，FIQ 被禁止。

② 状态控制位

T 位是处理器的状态控制位。

T=0，处理器处于 ARM 状态（即正在执行 32 位的 ARM 指令）。

T=1，处理器处于 Thumb 状态（即正在执行 16 位的 Thumb 指令）。

当然，T 位只有在 T 系列的 ARM 处理器上才有效，在非 T 系列的 ARM 版本中，T 位将始终为 0。

③ 模式控制位

M[4：0]作为位模式控制位，这些位的组合确定了处理器处于哪种状态。表 3.3 列出了其具体含义。

只有表中列出的组合是有效的，其他组合无效。

表 3.3 状态控制位 M[4：0]

M[4：0]	处理器模式	可以访问的寄存器
0b10000	User	PC，r14～r0，CPSR
0b10001	FIQ	PC，r14_fiq～r8_fiq，r7～r0，CPSR，SPSR_fiq
0b10010	IRQ	PC，r14_irq～r13_irq，r12～r0，CPSR，SPSR_irq
0b10011	Supervisor	PC，r14_svc～r13_svc，r12～r0，CPSR，SPSR_svc
0b10111	Abort	PC，r14_abt～r13_abt，r12～r0，CPSR，SPSR_abt
0b11011	Undefined	PC，r14_und～r13_und，r12～r0，CPSR，SPSR_und
0b11111	System	PC，r14～r0，CPSR(ARM v4 及更高版本)

注意 由于用户模式（User）和系统模式（System）是非异常模式，所以没有单独的 SPSR 保存程序状态字。在用户模式或系统模式下，读 SPSR 将返回一个不可预知的值，而写 SPSR 将被忽略。

3.4 异常中断处理

异常或中断是用户程序中最基本的一种执行流程和形态。这部分主要对 ARM 架构下的异常中断做详细说明。

ARM 有 7 种类型的异常，按优先级从高到低的排列如下：复位异常（Reset）、数据异常（Data Abort）、快速中断异常（FIQ）、外部中断异常（IRQ）、预取异常（Prefetch Abort）、软件中断(SWI)和未定义指令异常（Undefined instruction）。

> **注意** 在 ARM 文档中，使用术语 Exception 来描述异常。Exception 主要是从处理器被动接受异常的角度出发，而 Interrupt 带有向处理器主动申请的色彩。在本书中，对“异常”和“中断”不做严格区分，两者都是指请求处理器打断正常的程序执行流程，进入特定程序循环的一种机制。

3.4.1 异常种类

ARM 体系结构中，存在 7 种异常处理。当异常发生时，处理器会把 PC 设置为一个特定的存储器地址。这一地址放在被称为向量表（vector table）的特定地址范围内。向量表的入口是一些跳转指令，跳转到专门处理某个异常或中断的子程序。

存储器映射地址 0x00000000 是为向量表（一组 32 位字）保留的。在有些处理器中，向量表可以选择定位在存储空间的高地址（从偏移量 0xffff0000 开始）。一些嵌入式操作系统，如 Linux 和 Windows CE 就要利用这一特性。

表 3.4 列出了 ARM 的 7 种异常。

表 3.4　　ARM 的 7 种异常

异 常 类 型	处理器模式	执行低地址	执行高地址
复位异常（Reset）	特权模式	0x00000000	0xFFFF0000
未定义指令异常（Undefined interrupt）	未定义指令中止模式	0x00000004	0xFFFF0004
软中断异常（Software Abort）	特权模式	0x00000008	0xFFFF0008
预取异常（Prefetch Abort）	数据访问中止模式	0x0000000C	0xFFFF000C
数据异常（Data Abort）	数据访问中止模式	0x00000010	0xFFFF0010
外部中断请求 IRQ	外部中断请求模式	0x00000018	0xFFFF0018
快速中断请求 FIQ	快速中断请求模式	0x0000001C	0xFFFF001C

异常处理向量表如图 3.5 所示。

当异常发生时，分组寄存器 r14 和 SPSR 用于保存处理器状态，操作伪指令如下。

```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4：0] = exception mode number
CPSR[5] = 0   /*进入 ARM 状态*/
If <exception_mode> = = reset or FIQ then
   CPSR[6] = 1              /*屏蔽快速中断 FIQ*/
   CPSR[7] = 1              /*屏蔽外部中断 IRQ*/
   PC = exception vector address
```

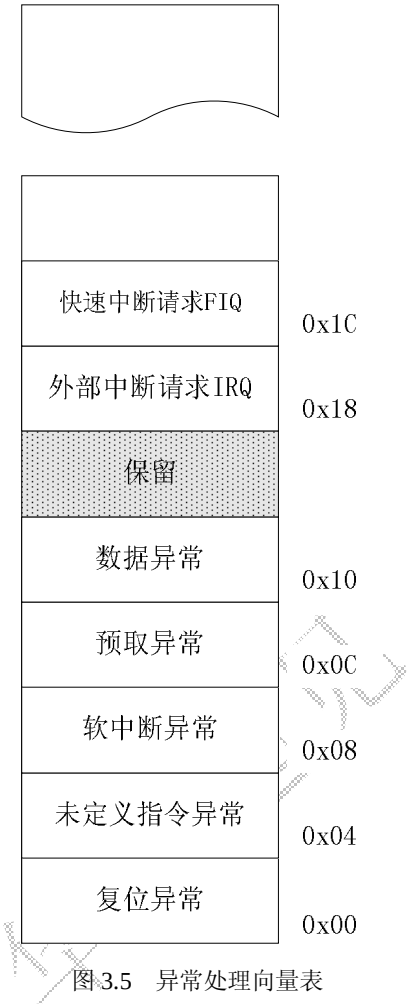


图 3.5　异常处理向量表

异常返回时，SPSR 内容恢复到 CPSR，连接寄存器 r14 的内容恢复到程序计数器 PC。

1．复位异常

当处理器的复位引脚有效时，系统产生复位异常中断，程序跳转到复位异常中断处理程序处执行。复位异常中断通常用在下面两种情况下。

- 系统上电。
- 系统复位。

当复位异常时，系统执行下列伪操作。

```
R14_svc = UNPREDICTABLE value
SPSR_svc = UNPREDICTABLE value
CPSR[4：0] = 0b10011        /*进入特权模式*/
CPSR[5] = 0                 /*处理器进入 ARM 状态*/
CPSR[6] = 1                 /*禁止快速中断*/
CPSR[7] = 1                 /*禁止外设中断*/
```

```
If high vectors configured then
   PC = 0xffff0000
Else
   PC = 0x00000000
```

复位异常中断处理程序将进行一些初始化工作，内容与具体系统相关。下面是复位异常中断处理程序的主要功能。

- 设置异常中断向量表。
- 初始化数据栈和寄存器。
- 初始化存储系统，如系统中的 MMU 等。
- 初始化关键的 I/O 设备。
- 使能中断。
- 处理器切换到合适的模式。
- 初始化 C 变量，跳转到应用程序执行。

2．未定义指令异常

当 ARM 处理器执行协处理器指令时，它必须等待一个外部协处理器应答后，才能真正执行这条指令。若协处理器没有相应，则发生未定义指令异常。

未定义指令异常可用于在没有物理协处理器的系统上，对协处理器进行软件仿真，或通过软件仿真实现指令集扩展。例如，在一个不包含浮点运算的系统中，CPU 遇到浮点运算指令时，将发生未定义指令异常中断，在该未定义指令异常中断的处理程序中可以通过其他指令序列仿真浮点运算指令。

仿真功能可以通过下面步骤实现。

① 将仿真程序入口地址链接到向量表中未定义指令异常中断入口处（0x00000004 或 0xffff0004），并保存原来的中断处理程序。

② 读取该未定义指令的 bits[27：24]，判断其是否是一条协处理器指令。如果 bits[27：24]值为 0b1110 或 0b110x，该指令是一条协处理器指令；否则，由软件仿真实现协处理器功能，可以同过 bits[11：8]来判断要仿真的协处理器功能（类似于 SWI 异常实现机制）。

③ 如果不仿真该未定义指令，程序跳转到原来的未定义指令异常中断的中断处理程序执行。

当未定义异常发生时，系统执行下列的伪操作。

```
r14_und = address of next instruction after the undefined instruction
SPSR_und = CPSR
CPSR[4：0] = 0b11011          /*进入未定义指令模式*/
CPSR[5] = 0                   /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1                   /*禁止外设中断*/
If  high vectors configured then
     PC = 0xffff0004
```

```
Else
     PC = 0x00000004
```

3．软中断 SWI

软中断异常发生时，处理器进入特权模式，执行一些特权模式下的操作系统功能。软中断异常发生时，处理器执行下列伪操作。

```
r14_svc = address of next instruction after the SWI instruction
SPSR_und = CPSR
CPSR[4：0] = 0b10011          /*进入特权模式*/
CPSR[5] = 0                   /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1                   /*禁止外设中断*/
If  high vectors configured then
     PC = 0xffff0008
Else
     PC = 0x00000008
```

4．预取指令异常

预取指令异常使由系统存储器报告的。当处理器试图去取一条被标记为预取无效的指令时，发生预取异常。

如果系统中不包含 MMU 时，指令预取异常中断处理程序只是简单地报告错误并退出。若包含 MMU，引起异常的指令的物理地址被存储到内存中。

预取异常发生时，处理器执行下列伪操作。

```
r14_svc = address of the aborted instruction + 4
SPSR_und = CPSR
CPSR[4：0] = 0b10111           /*进入特权模式*/
CPSR[5] = 0                    /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1                    /*禁止外设中断*/
If  high vectors configured then
    PC = 0xffff000C
Else
    PC = 0x0000000C
```

5．数据访问中止异常

数据访问中止异常是由存储器发出数据中止信号，它由存储器访问指令 Load/Store 产生。当数据访问指令的目标地址不存在或者该地址不允许当前指令访问时，处理器产生数据访问

中止异常。

当数据访问中止异常发生时，处理器执行下列伪操作。

```
r14_abt = address of the aborted instruction + 8
SPSR_abt = CPSR
CPSR[4：0] = 0b10111
CPSR[5] = 0
/*CPSR[6]保持不变*/
CPSR[7] = 1            /*禁止外设中断*/
If  high vectors configured then
    PC = 0xffff000C10
Else
    PC = 0x00000010
```

当数据访问中止异常发生时，寄存器的值将根据以下规则进行修改。

① 返回地址寄存器 r14 的值只与发生数据异常的指令地址有关，与 PC 值无关。

② 如果指令中没有指定基址寄存器回写，则基址寄存器的值不变。

③ 如果指令中指定了基址寄存器回写，则寄存器的值和具体芯片的 Abort Models 有关，由芯片的生产商指定。

④ 如果指令只加载一个通用寄存器的值，则通用寄存器的值不变。

⑤ 如果是批量加载指令，则寄存器中的值是不可预知的值。

⑥ 如果指令加载协处理器寄存器的值，则被加载寄存器的值不可预知。

6．外部中断 IRQ

当处理器的外部中断请求引脚有效，而且 CPSR 寄存器的 I 控制位被清除时，处理器产生外部中断 IRQ 异常。系统中各外部设备通常通过该异常中断请求处理器服务。

当外部中断 IRQ 发生时，处理器执行下列伪操作。

```
r14_irq = address of next instruction to be executed + 4
SPSR_irq = CPSR
CPSR[4：0] = 0b10010          /*进入特权模式*/
CPSR[5] = 0                   /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1                   /*禁止外设中断*/
If  high vectors configured then
    PC = 0xffff0018
Else
    PC = 0x00000018
```

7．快速中断 FIQ

当处理器的快速中断请求引脚有效且 CPSR 寄存器的 F 控制位被清除时，处理器产生快速中断请求 FIQ 异常。

当快速中断异常发生时，处理器执行下列伪操作。

```
r14_fiq = address of next instruction to be executed + 4
SPSR_fiq = CPSR
CPSR[4：0] = 0b10001          /*进入 FIQ 模式*/
CPSR[5] = 0
CPSR[6] = 1
CPSR[7] = 1
If  high vectors configured then
    PC= 0xffff001c
Else
    PC = 0x0000001c
```

3.4.2 异常优先级

每一种异常按表 3.5 中设置的优先级得到处理。

表 3.5 异常优先级

优 先 级		异 常
最高	1	复位异常
	2	数据中止
	3	快速中断请求
	4	中断请求
	5	预取指令异常
	6	软件中断
最低	7	未定义指令

异常可以同时发生，处理器按表 3.5 的优先级顺序处理异常。例如，复位异常的优先级最高，处理器上电时发生复位异常。所以当产生复位时，它将优先于其他异常得到处理。同样，当一个数据访问中止异常发生时，它将优先于除复位异常外的其他所有异常。

优先级最低的 2 种异常是软件中断和未定义指令异常。因为正在执行的指令不可能既是一条 SWI 指令，又是一条未定义指令，所以软件中断异常 SWI 和未定义指令异享有相同的优先级。

3.4.3 处理器模式和异常

每一种异常都会导致内核进入一种特定的模式。表 3.6 显示了 ARM 处理器异常及其对应的模式。此外，也可以通过编程改变 CPSR，进入任何一种 ARM 处理器模式。

注意 用户和系统模式是仅有的不可通过异常进入的两种模式，也就是说，要进入这两种模式，必须通过编程改变 CPSR。

表 3.6 ARM 处理器异常及其对应模式

异　常	模　式	用　途
快速中断请求	FIQ	进行快速中断请求处理
外部中断请求	IRQ	进行外部中断请求处理
SWI	SVC	进行操作系统的高级处理
复位	SVC	进行操作系统的高级处理
预取指令中止异常	ABORT	虚存和存储器保护
数据中止异常	ABORT	虚存和存储器保护
未定义指令	Undefined	软件模拟硬件协处理器

3.4.4 异常响应流程

1. 判断处理器状态

当异常发生时，处理器自动切换到 ARM 状态，所以在异常处理函数中要判断在异常发生前处理器是 ARM 状态还是 Thumb 状态。这可以通过检测 SPSR 的 T 位来判断。

通常情况下，只有在 SWI 处理函数中才需要知道异常发生前处理器的状态。所以在 Thumb 状态下，调用 SWI 软中断异常必须注意以下两点。

① 发生异常的指令地址为（lr－2）而不是（lr－4）。

② Thumb 状态下的指令是 16 位的，在判断中断向量号时使用半字加载指令 LDRH。

下面的例子显示了一个标准的 SWI 处理函数，在函数中通过 SPSR 的 T 位判断异常发生前的处理器状态。

```
    T_bit EQU 0x20              ; bit 5. SPSR 中的 ARM/Thumb 状态位,
    :
    :
    SWIHandler
    STMFD sp!, {r0-r3,r12,lr}   ; 寄存器压栈，保护程序现场
    MRS r0, spsr                ; 读 SPSR 寄存器，判断异常发生前的处理器状态
    TST r0, #T_bit              ; 检测 SPSR 的 T 位，判断异常发生前是否为 Thumb 状态
    LDRNEH r0,[lr,#-2]          ; 如果是 Thumb 状态,使用半字加载指令读取发生异常的指令地
址
    BICNE r0,r0,#0xFF00         ; .提取中断向量号.
    LDREQ r0,[lr,#-4]           ; 如果是 ARM 状态，使用字加载指令，读取发生异常的指令地址
    BICEQ r0,r0,#0xFF000000     ; 提取中断向量号并将中断向量号存入 r0
    ; r0 存储中断向量号
```

```
CMP r0, #MaxSWI             ; 判断中断是否超出范围
LDRLS pc, [pc, r0, LSL#2]   ; 如果未超出范围，跳转到软中断向量表 Switable
B SWIOutOfRange             ; 如果超出范围，跳转到软中断越界处理程序
switable
DCD do_swi_1
DCD do_swi_2
:
:
do_swi_1
; 1 号软中断处理函数
LDMFD sp!, {r0-r3,r12,pc}^ ; Restore the registers and return.
                           ; 恢复寄存器并返回
do_swi_2                       ; 2 号软中断处理函数
:
```

2．向量表

如前面介绍向量表时提到的，每一个异常发生时总是从异常向量表开始跳转。最简单的一种情况是向量表里面的每一条指令直接跳向对应的异常处理函数。其中快速中断处理函数 FIQ_handler（）可以直接从地址 0x1C 处开始，省下一条跳转指令，如图 3.6 所示。

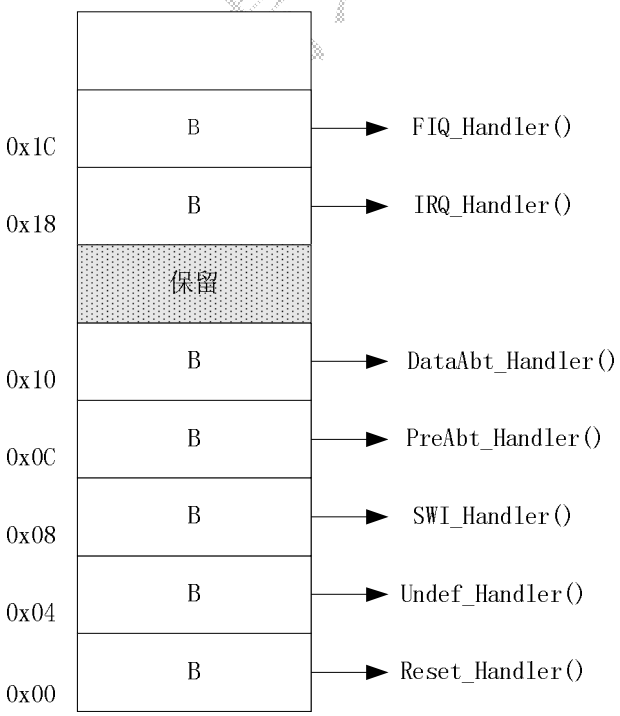


图 3.6　异常处理向量表

但跳转指令 B 的跳转范围为±32MB，但很多情况下不能保证所有的异常处理函数都定位在向量的 32MB 范围内，需要更大范围的跳转，而且由于向量表空间的限制，只能由一条指令完成。具体实现方法有下面两种。

（1）MOV　PC，＃imme_value

这种办法将目标地址直接赋值给 PC。但这种方法受格式限制不能处理任意立即数。这个立即数由一个 8 位数值循环右移偶数位得到。

（2）LDR　PC，[PC+offset]

把目标地址先存储在某一个合适的地址空间，然后把这个存储器单元的 32 位数据传送给 PC 来实现跳转。

这种方法对目标地址值没有要求。但是存储目标地址的存储器单元必须在当前指令的±4KB 空间范围内。

注意　在计算指令中引用 offset 数值的时候，要考虑处理器流水线中指令预取对 PC 值的影响。

3.4.5　从异常处理程序中返回

当一个异常处理返回时，一共有 3 件事情需要处理：通用寄存器的恢复、状态寄存器的恢复以及 PC 指针的恢复。通用寄存器的恢复采用一般的堆栈操作指令即可，下面重点介绍状态寄存器的恢复以及 PC 指针的恢复。

1．恢复被中断程序的处理器状态

PC 和 CPSR 的恢复可以通过一条指令来实现，下面是 3 个例子。

- MOVS　PC，LR
- SUBS　PC，LR，＃4
- LDMFD　SP!，{PC}^

这几条指令是普通的数据处理指令，特殊之处在于它们把程序计数器寄存器 PC 作为目标寄存器，并且带了特殊的后缀“S”或“^”。其中“S”或“^”的作用就是使指令在执行时，同时完成从 SPSR 到 CPSR 的拷贝，达到恢复状态寄存器的目的。

2．异常的返回地址

异常返回时，另一个非常重要的问题就是返回地址的确定。前面提到过，处理器进入异常时会有一个保存 LR 的动作，但是该保持值并不一定是正确中断的返回地址。以一个简单的指令执行流水状态图来对此加以说明，如图 3.7 所示。

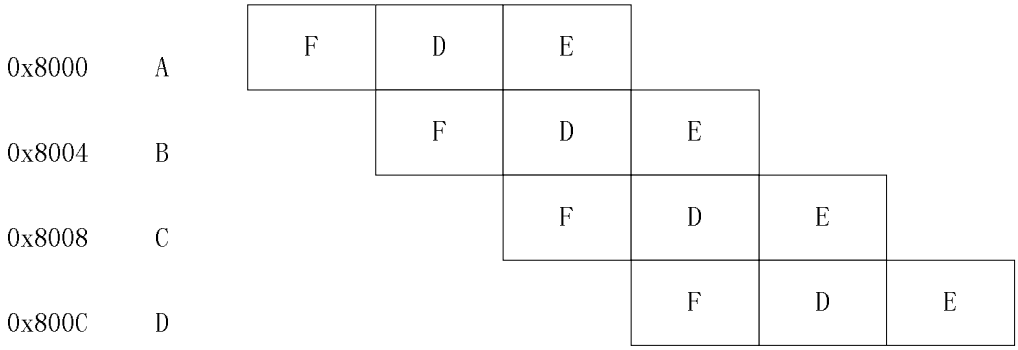


图 3.7 3 级流水线示例

在 ARM 架构里，PC 值指向当前执行指令地址加 8。也就是说，当执行指令 A（地址 0x8000）时，PC 等于 0x8000+8=0x8008，即等于指令 C 的地址。假设指令 A 是 BL 指令，则当执行时，会把 PC 值（0x8008）保存到 LR 寄存器。但是，接下来处理器会对 LR 进行一次自动调整，使 LR=LR－0x4。所以，最终保存在 LR 里面的是图 3.5 中所示的 B 指令地址。所以当从 BL 返回时，LR 里面正好是正确的返回地址。

同样的跳转机制在所有的 LR 自动保存操作中都存在。当进入中断响应时，处理器对保存的 LR 也进行一次自动调整，并且跳转动作也是 LR=LR－0x04。由此，就可以对不同异常类型的返回地址依次比较。

假设在指令 B 处（地址 0x8004）发生了异常，进入异常相应后，LR 经过跳转保存的地址值应该是 C 的地址 0x8008。

（1）软中断异常

如果发生软中断异常，即指令 B 为 SWI 指令，从 SWI 中断返回后下一条执行指令就是 C，正好是 LR 寄存器保存的地址，所以只有直接把 LR 恢复给 PC 即可。

（2）IRQ 或 FIQ 异常

如果发生的是 IRQ 或 FIQ 异常，因为外部中断请求中断了正在执行的指令 B，当中断返回后，需要重新回到 B 指令执行，也就是说，返回地址应该是 B（0x8004），需要把 LR 减 4 送 PC。

（3）Data Abort 数据中止异常

在指令 B 处进入数据异常的相应，但导致数据异常的原因却应该是上一条指令 A。当中断处理程序恢复数据异常后，要回到 A 重新执行导致数据异常的指令，因此返回地址应该是 LR 加 8。

为方便起见，表 3.7 总结了各异常和返回地址的关系

表 3.7 异常和返回地址

异　常	地　址	用　途
复位	—	复位没有定义 LR
数据中止	LR－8	指向导致数据中止异常的指令
FIQ	LR－4	指向发生异常时正在执行的指令

IRQ	LR－4	指向发生异常时正在执行的指令
预取指令中止	LR－4	指向导致预取指令异常的那条指令
SWI	LR	执行 SWI 指令的下一条指令
未定义指令	LR	指向未定义指令的下一条指令

3.4.6 在应用程序中安装异常处理程序

1．使用汇编语言安装异常处理程序

如果系统启动不依赖于 Debug 或 Debug monitor 软件，可以使用汇编语言在系统启动时直接安装异常处理程序。

下面的例子显示了系统从 0x0 地址启动，直接安装异常处理程序的方法。

```
Vector_Init_Block
    LDR PC, Reset_Addr
    LDR PC, Undefined_Addr
    LDR PC, SWI_Addr
    LDR PC, Prefetch_Addr
    LDR PC, Abort_Addr
    NOP                         ;保留向量
    LDR PC, IRQ_Addr
    LDR PC, FIQ_Addr
Reset_Addr DCD Start_Boot
Undefined_Addr DCD Undefined_Handler
SWI_Addr DCD SWI_Handler
Prefetch_Addr DCD Prefetch_Handler
Abort_Addr DCD Abort_Handler
    DCD 0                       ;保留向量
IRQ_Addr DCD IRQ_Handler
FIQ_Addr DCD FIQ_Handler
```

有些情况下，系统 0x0 地址不一定是 ROM。如果 0x0 地址为 RAM，那么就系统将中断向量表从 ROM 复制 RAM，下面的例子显示了这样一个过程。

```
MOV R8, #0
ADR R9, Vector_Init_Block
LDMIA R9!,{r0-r7}               ;复制中断向量表 (8 words)
STMIA R8!,{r0-r7}
LDMIA R9!,{r0-r7}               ;复制由伪操作 DCD 定义的地址
STMIA R8!,{r0-r7}
```

注意 可以使用 Scatter 文件定义加载向量表的地址，这样上述代码的拷贝工作由 C 库函数完成。

2．使用 C 语言安装异常处理程序

程序中有时需要在 main()函数中使用 C 语言安装中断向量表。这就要求指令经编译后的解码能安装在内存的正确位置。

（1）向量表中使用跳转指令的情况

如果在向量表中使用跳转指令，使用下面的步骤完成向量表的安装。

① 读取异常处理程序的地址。

② 从异常处理程序地址中减去向量表中的偏移。

③ 为适应指令流水线，将上一步得到的地址减 8。

④ 将得到的结果右移 2 位，得到以字为单位的地址偏移量。

⑤ 将结果的高 8 位清零，得到跳转指令的 24 位偏移量。

⑥ 将上一步得到的结果和 0xea000000（无条件跳转指令编码）做逻辑与操作，从而得到要写到向量表中的跳转指令的正确编码。

下面的例子显示了这样一个标准过程。

```
unsigned Install_Handler (unsigned routine, unsigned *vector)
{ unsigned vec, oldvec;
vec = ((routine - (unsigned)vector - 0x8)>>2);
if ((vec & 0xFF000000))
{
/* diagnose the fault */
prinf ("Installation of Handler failed");
exit (1);
}
vec = 0xEA000000 | vec;
oldvec = *vector;
*vector = vec;
return (oldvec);
}
```

（2）在向量表中使用加载 PC 指令

在向量表中使用加载 PC 指令，按照下面的步骤完成。

① 读取异常处理程序地址。

② 从异常处理程序地址中减去向量表中的偏移。

③ 为适应指令流水线，将上一步得到的地址减 8。

④ 保留结果的后 12 位。

⑤ 将结果与 0xe59ff000（LDR PC, [PC,#offset]）做逻辑或操作，从而得到要写到向量表

中的跳转指令的正确编码。

⑥ 将异常处理程序的地址放到相应的存储单元。

下面的例子显示了一个标准的 C 语言过程。

```
unsigned Install_Handler (unsigned location, unsigned *vector)
{ unsigned vec, oldvec;
vec = ((unsigned)location - (unsigned)vector - 0x8) | 0xe59ff000;
oldvec = *vector;
*vector = vec;
return (oldvec);
}
```

3.4.7 FIQ 和 IRQ 中断处理函数的设计

1. 中断分支

ARM 内核只有两个外部中断输入信号 nFIQ 和 nIRQ。但对于一个系统来说，中断源可能多达几十个。为此，在系统集成的时候，一般都会有一个异常控制器来处理异常信号，如图 3.8 所示。

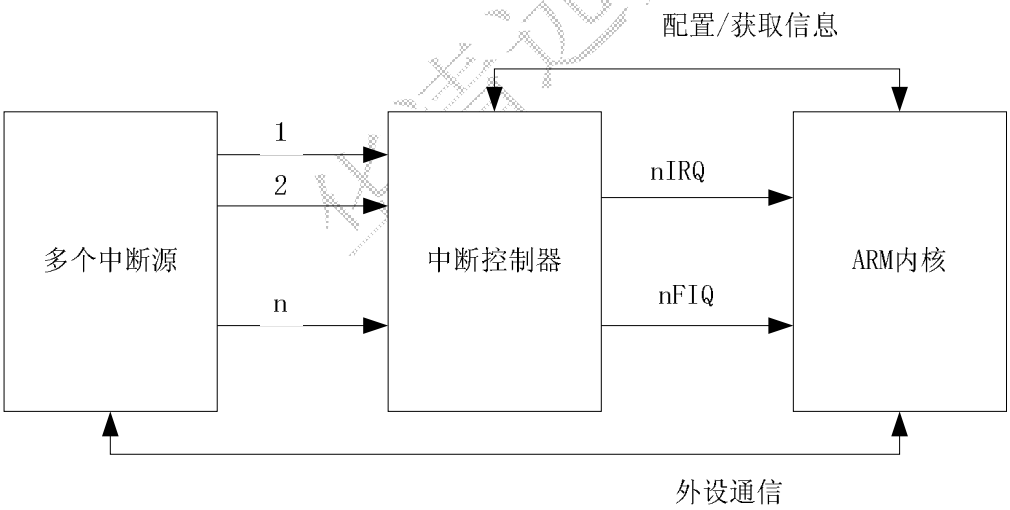


图 3.8 中断系统

这时候用户程序可能存在多个 IRQ/FIQ 的中断处理函数。为了使从向量表开始的跳转始终能找到正确的处理函数入口，需要设置一套处理机制和方法。

多数情况下是由软件来处理异常分支的，因为软件可以通过读取中断控制器来获得中断源的信息，如图 3.9 所示。

有些芯片可能支持特殊的硬件分支功能，这需要查看具体的芯片说明。

因为软件的灵活性，可以设计出比图 3.9 更好的流程控制方法，如图 3.10 所示。

Int_vector_table 是用户自己开辟的一块存储器空间，里面按次序存放异常处理函数的地

址。IRQ_Handler()从中断控制器获取中断源信息，然后再从 Int_vector_table 中的对应地址单元得到异常处理函数的入口地址，完成一次异常响应的跳转。这种方法的好处是用户程序在运行过程中，能够很方便地动态改变异常服务内容。

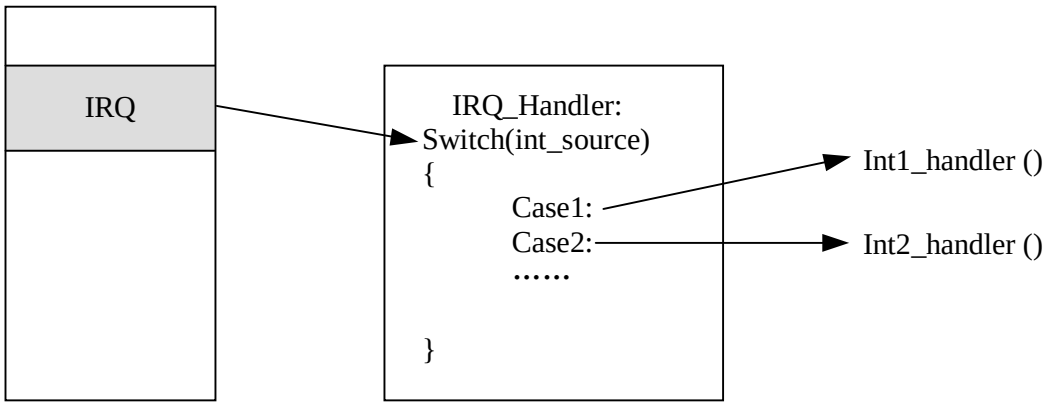


图 3.9　软件控制中断分支

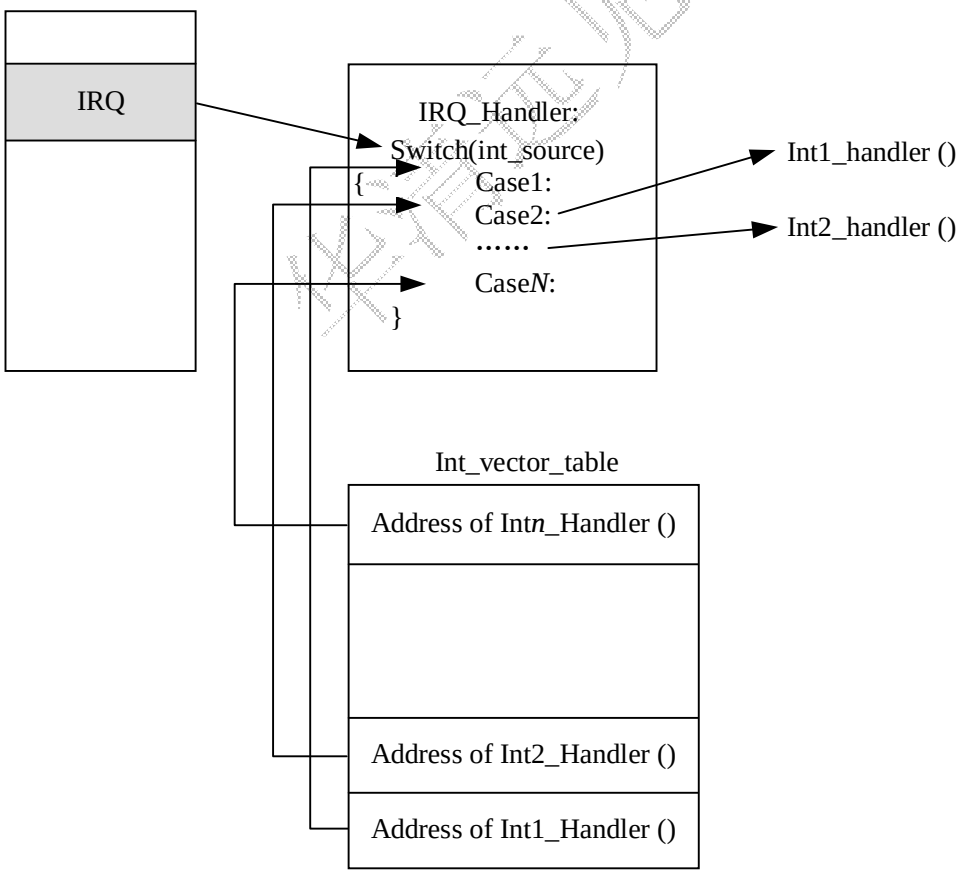


图 3.10　灵活的软件分支设计

进入异常处理程序后，用户可以完全按照自己的意愿来进行程序设计，包括调用 Thumb 状态的函数等。但对于绝大多数的系统来说，有两个步骤必须处理，一是现场保护，二是要把中断控制器中对应的中断状态标识清除，表明该中断请求已经得到响应，否则，中断函数退出以后，又会被再一次触发，从而进入周而复始的死循环。

2．ARM 编译器对中断处理函数编写的扩展

考虑到中断处理函数在现场保护和返回地址的处理上与普通函数的不同之处，不能直接把普通函数体连接到异常向量表上，需要在上面加上一层封装，下面是一个例子。

```
IRQ_Handler                         ;中断相应函数
    STMFD    SP!,{r0-r12,lr}        ;保护现场，一般只需要保护{r0-r3,LR}
    BL   IrqHandler                 ;进入普通处理函数，C 或汇编均可
……
    LDMFD    sp!,{r0-r12,LR}        ;恢复现场
    SUBS    pc,lr,#4                ;中断返回，注意返回地址
```

为了方便使用高级语言直接编写异常处理函数，ARM 编译器对此做了特定的扩展，可以使用函数声明关键字_irq，这样编译出来的函数就可以满足异常响应对现场保护和恢复的需要，并且自动加入 LR 减 4 的处理，符合 IQR 和 FIQ 中断处理的要求。

下面的例子显示了使用_irq 对中断处理函数产生的影响。

C 语言源程序如下。

```
__irq void IRQHandler (void)
{
volatile unsigned int *base = (unsigned int *) 0x80000000;
if (*base == 1)
{
/*调用 C 语言中断处理函数*/
C_int_handler();
}
/*清楚中断标志*/
*(base+1) = 0;
}
```

使用 armcc 编译出的汇编代码如下。

```
IRQHandler PROC
    STMFD sp!,{r0-r4,r12,lr}
    MOV r4,#0x80000000
    LDR r0,[r4,#0]
    SUB sp,sp,#4
    CMP r0,#1
```

```
    BLEQ C_int_handler
    MOV r0,#0
    STR r0,[r4,#4]
    ADD sp,sp,#4
    LDMFD sp!,{r0-r4,r12,lr}
    SUBS pc,lr,#4
    ENDP
```

如果不使用_irq 子程序声明关键字，编译出的汇编代码如下。

```
IRQHandler PROC
    STMFD sp!,{r4,lr}
    MOV r4,#0x80000000
    LDR r0,[r4,#0]
    CMP r0,#1
    BLEQ C_int_handler
    MOV r0,#0
    STR r0,[r4,#4]
    LDMFD sp!,{r4,pc}
    ENDP
```

3．可重入中断设计

在缺省情况下，ARM 中断是不可重入的。因为一旦进入异常响应状态，ARM 自动关闭中断使能。如果在异常处理过程中，简单地打开中断使能而发生中断嵌套时，显然新的异常处理将破坏原来的中断现场而导致出错。但有时需要中断必须是可重入的，因此要通过程序设计来解决这个问题。其中有两个关键问题。

① 新中断使能之前，必须要保护好前一个中断的现场信息。比如 LR_irq 和 SPSR_irq 等，这一点比较容易做的。

② 中断处理过程中对 BL 进行保护。

在中断处理函数中发生函数调用 BL 是很常见的，假设有下面一种情况。

```
IRQ_Handler:
    ……
    BL    Foo
    ADD
```

其中，

```
Foo:
    STMFD  SP!,{r0-r3，LR}
    ……
```

```
    LDMFD  SP!{r0-r3，PC}
```

上述程序，在 IRQ 处理函数 IRQ_Handler()中调用了函数 Foo()。若是在 IRQ_Handler()里面中断可重入的话，可能发生问题，考察下面的情况：当新的中断请求恰好在“BL　Foo”指令执行完成后发生。这时候 LR_irq 寄存器（因在 IRQ 模式下，所以是 LR_irq）的值将调整为 BL 指令的下一条指令（ADD）地址，使其能从 Foo()正确返回；但是因为这时候发生了中断请求，接下来要进行新中断的响应，处理器在新中断响应过程中也要进行 LR_irq 保存。这次对 LR_irq 的操作发生了冲突，当新中断返回后，往下执行 STMFD 指令，这时候压栈的 LR 已不是原来的 ADD 指令地址，从而使子程序 Foo()无法正确返回。

这个问题无法通过增加额外的现场保护指令来解决。一个办法就是在重新使能中断之前改变处理器模式，也就是使上面程序的“BL　Foo”指令不要运行在 IRQ 模式下。这样当新的中断发生时，就不会造成 LR 寄存器的冲突。考虑 ARM 的所有运行模式，采用 SYSTEM 模式是比较合适的，因为它是特权模式，不是 IRQ 模式，与中断响应无关。

下面的例子显示了标准的 IRQ/FIQ 异常中断处理程序。

```
    PRESERVE8
    AREA INTERRUPT, CODE, READONLY
    IMPORT C_irq_handler
IRQ
    SUB lr, lr, #4              ;跳转返回地址
STMFD sp!, {lr}                 ;保存返回地址
MRS r14, SPSR                   ;读取 SPSR
STMFD sp!, {r12, r14}           ;保存寄存器
; 清除中断源
MSR CPSR_c, #0x1F               ;切换到 SYSTEM 模式,
STMFD sp!, {r0-r3, lr}          ;保存 lr_USR 和其他使用到的寄存器
BL C_irq_handler                ;跳转到 C 中断处理函数
LDMFD sp!, {r0-r3, lr}          ;恢复用户模式寄存器
MSR CPSR_c, #0x92               ;切换回 irq 模式
LDMFD sp!, {r12, r14}
MSR SPSR_cf, r14
LDMFD sp!, {pc}^
END
```

3.4.8　SWI 异常处理函数的设计

本小节主要介绍编写 SWI 处理程序时需要注意的几个问题，包括下面内容。

- 判断 SWI 中断号。
- 使用汇编语言编写 SWI 异常处理函数。
- 使用 C 语言编写 SWI 异常处理函数。
- 在特权模式下使用 SWI 异常中断处理。

- 从应用程序中调用 SWI。
- 从应用程序中动态调用 SWI。

1．判断 SWI 中断号

当发生 SWI 异常，进入异常处理程序时，异常处理程序必须提取 SWI 中断号，从而得到用户请求的特定 SWI 功能。

在 SWI 指令的编码格式中，后 24 位称为指令的“comment field”。该域保存的 24 位数，即为 SWI 指令的中断号，如图 3.11 所示。

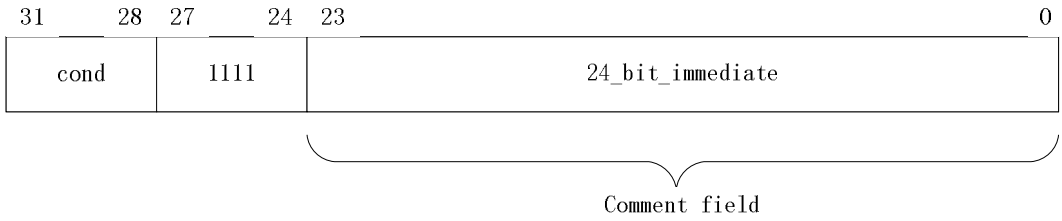


图 3.11　SWI 指令编码格式

第一级的 SWI 处理函数通过 LR 寄存器内容得到 SWI 指令地址，并从存储器中得到 SWI 指令编码。通常这些工作通过汇编语言、内嵌汇编来完成。

下面的例子显示了提取中断向量号的标准过程。

```
PRESERVE8
AREA TopLevelSwi, CODE, READONLY    ;第一级 SWI 处理函数.
EXPORT SWI_Handler
SWI_Handler
STMFD sp!,{r0-r12,lr}               ;保存寄存器
LDR r0,[lr,#-4]                     ;计算 SWI 指令地址.
BIC r0,r0,#0xff000000               ;提取指令编码的后 24 位
;
; 提取出的中断号放 r0 寄存器，函数返回
;
LDMFD sp!, {r0-r12,pc}^             ;恢复寄存器
END
```

例子中，使用 LR-4 得到 SWI 指令的地址，再通过“BIC r0, r0, #0xFF000000”指令提取 SWI 指令中断号。

2．汇编语言编写 SWI 异常处理函数

最简单的方法是利用得到的中断向量号，使用跳转表直接跳转到实现相应 SWI 功能的处理程序。

下面的例子，使用汇编语言实现了这种跳转。

```
    CMP r0,#MaxSWI              ;中断向量范围检测
    LDRLS pc, [pc,r0,LSL #2]
    B SWIOutOfRange
SWIJumpTable
    DCD SWInum0
    DCD SWInum1
; 使用 DCD 定义各功能函数入口地址
SWInum0                         ;0 号中断
    B EndofSWI
SWInum1                         ;1 号中断
    B EndofSWI
;
EndofSWI
```

3．使用 C 语言编写 SWI 异常处理函数

虽然第一级 SWI 处理函数（完成中断向量号的提取）必须用汇编语言完成，但第二级中断处理函数（根据提取的中断向量号，跳转到具体处理函数）就可以使用 C 语言来完成。

因为第一级的中断处理函数已经将中断号提取到寄存器 r0 中，所以根据 AAPCS 函数调用规则，可以直接使用 BL 指令跳转到 C 语言函数，而且中断向量号作为第一个参数被传递到 C 函数。

例如汇编中使用了“BL　C_SWI_Handler”跳转到 C 语言的第二级处理函数，则第二级的 C 语言函数示例如下所示。

```
void C_SWI_handler (unsigned number)
{
    switch (number)
     {
      case 0 : /* SWI number 0 code */
      break;
      case 1 : /* SWI number 1 code */
      break;
      ...
      default : /* Unknown SWI - report error */
     }
}
```

另外，如果需要传递的参数多于 1 个，那么可以使用堆栈，将堆栈指针作为函数的参数传递给 C 类型的二级中断处理程序，就可以实现在两级中断之间传递多个参数。

例如：

```
MOV r1, sp                ;将传递的第二个参数（堆栈指针）放到 r1 中
BL C_SWI_Handler          ;调用 C 函数
```

相应的 C 函数的入口变为：

```
void C_SWI_handler(unsigned number, unsigned *reg)
```

同时，C 函数也可以通过堆栈返回操作的结果。

4．在特权模式下使用 SWI 异常处理

在特权模式下使用 SWI 异常处理，和 IRQ/FIQ 中断嵌套基本类似。当执行 SWI 指令后，处理器执行下面操作。

① 处理器进入特权模式。

② 将程序状态字内容 CPSR 保存到 SPSR_svc。

③ 返回地址放入 LR_svc。

如果处理器已经处于特权模式，再发生 SWI 异常，则 LR_svc 和 SPSR_svc 寄存器的值将丢失。

所以在特权模式下，调用 SWI 软中断异常，必须先将 LR_svc 和 SPSR_svc 寄存器的值压栈保护。下面的例子显示了一个可以在特权模式下调用的 SWI 处理函数。

```
      AREA SWI_Area, CODE, READONLY
    PRESERVE8
    EXPORT SWI_Handler
    IMPORT C_SWI_Handler
                    T_bit EQU 0x20
SWI_Handler
    STMFD sp!,{r0-r3,r12,lr}          ;寄存器压栈保护
    MOV r1, sp                        ;堆栈指针放 r1 作为参数传递.
    MRS r0, spsr                      ;读取 spsr.
    STMFD sp!, {r0, r3}               ;将 spsr 压栈保护
    ;
    ;
    LDR r0,[lr,#-4]                   ;计算 SWI 指令地址.
    BIC r0,r0,#0xFF000000             ;读取 SWI 中断向量号.
    ; r0 存放中断向量号
    ; r1 堆栈指针
    BL C_SWI_Handler                  ;调用 C 程序的 SWI 处理函数.
    LDMFD sp!, {r0, r3}               ;从堆栈中读取 spsr.
```

```
    MSR spsr_cf, r0                    ;恢复 spcr
    LDMFD sp!, {r0-r3,r12,pc}^         ;恢复其他寄存器并返回.
    END
```

5．从应用程序中调用 SWI

可从汇编语言或 C/C++ 中调用 SWI。

（1）从汇编应用程序中调用 SWI

从汇编语言程序中调用 SWI，只要遵循 AAPCS 标准即可。调用前，设定所有必须的值并发出相关的 SWI。例如：

```
MOV r0, #65       ; 将软中断的子功能号放到 r0 中
SWI 0x0
```

注意 SWI 指令和其他所以 ARM 指令一样，可以被条件执行。

（2）从 C 应用程序中调用 SWI

在 C 或 C++应用程序中调用 SWI，要将 C 语言的子程序用编译器扩展_swi 声明，例如：

```
__swi(0) void my_swi(int);
……
……
……
my_swi(65);
```

编译器扩展_swi 确保了 SWI 以内联方式进行编译，而没有额外的开销。但有如下的 AAPCS 限制。

- 函数调用参数只能使用 r0～r3 传递。
- 函数返回值只能通过 r0～r3 传递。

向内联的 SWI 函数传递参数和向实际的子函数传递参数基本类似。但返回值的情况比较复杂。如果有两到四个返回值，则必须告诉编译程序返回值是以结构形式返回的，并使用 __value_in_regs 伪操作声明。这是因为基于结构值的函数通常被处理为一个 void（空）型函数，且第一个自变量必须为存放结果结构的地址。

下面的例子显示了对编号为 0x0、0x1、0x2 和 0x3 的 SWI 软中断的调用。其中，SWI0x0 和 SWI0x1 传递两个整型参数并返回一个单一结果；SWI0x2 传递 4 个参数并返回一个单一结果；而 SWI0x3 传递 4 个参数并通过结构体返回 4 个结果。

```
#include <stdio.h>
#include "swi.h"
unsigned *swi_vec = (unsigned *)0x08;
extern void SWI_Handler(void);
int main( void )
```

```
    {
    int result1, result2;
    struct four_results res_3;
    Install_Handler( (unsigned) SWI_Handler, swi_vec );
    printf("result1 = multiply_two(2,4) = %d\n", result1 = multiply_two(2,4));
    printf("result2 = multiply_two(3,6) = %d\n", result2 = multiply_two(3,6));
    printf("add_two( result1, result2 ) = %d\n", add_two( result1, result2 ));
    printf("add_multiply_two(2,4,3,6) = %d\n", add_multiply_two(2,4,3,6));
    res_3 = many_operations( 12, 4, 3, 1 );
    printf("res_3.a = %d\n", res_3.a );
    printf("res_3.b = %d\n", res_3.b );
    printf("res_3.c = %d\n", res_3.c );
    printf("res_3.d = %d\n", res_3.d );
    return 0;
    }
    __swi(0) int multiply_two(int, int);
    __swi(1) int add_two(int, int);
    __swi(2) int add_multiply_two(int, int, int, int);
    struct four_results
    {
    int a;
    int b;
    int c;
    int d;
    };
    __swi(3) __value_in_regs struct four_results many_operations(int, int, int,
int);
```

（3）应用程序中动态调用 SWI

在某些情形下，需要调用直到运行时才会知道其编号的 SWI。例如，当有很多相关操作可在同一目标上执行，并且每一个操作都有其自己的 SWI 时，就会发生这种情况。在此情况下，上一小节的方法不适用。

解决的方法有两种。

- 在运行时得到 SWI 功能号，然后构造出相应的 SWI 指令的编码，将该编码保存在某个存储单元中，将 PC 指针指向该单元，执行指令。
- 使用一个通用的 SWI 异常中断处理程序，将运行时需要调用的 SWI 功能号作为参数传递给该通用的 SWI 异常处理程序，通用的 SWI 异常中断处理程序根据参数值调用相应的 SWI 处理程序完成需要的操作。

通过汇编语言可以实现第二种解决办法：通过寄存器（通常为 r0 或 r12）传递所需要的操作数，这样可以重新编写 SWI 处理程序，对相应寄存器中的值进行处理。

但有些情况下，为了节省程序开销，需要直接使用 SWI 中断号对程序调用。例如，操作系统可能会使用单一的一条 SWI 指令并用寄存器来传递所需运算的编号。这使得其他 SWI 空间可用于特定应用程序的 SWI。在一个特定的应用程序中，如果从指令中提取 SWI 编号的开销太大，就可使用这个方法。ARM（0x123456）和 Thumb（0xAB）半主机方式的 SWI 就是这样实现的。

下面的例子显示了如何使用_swi 将 C 函数调用映射到半主机方式的 SWI。

```
#ifdef __thumb
/* Thumb 状态的 Semihosting 软中断处理*/
#define SemiSWI 0xAB
#else
/* ARM 状态下的 Semihosting 的软中断处理*/
#define SemiSWI 0x123456
#endif
/* 使用 Semihosting 软中断输出一个字符*/
__swi(SemiSWI) void Semihosting(unsigned op, char *c);
#define WriteC(c) Semihosting (0x3,c)
void write_a_character(int ch)
{
char tempch = ch;
WriteC( &tempch );
}
```

编译程序含有一个机制，用以支持使用 r12 来传递所需运算的值。根据 AAPCS 标准，r12 为 IP 寄存器，并且专用于函数调用。其他时间内可将其用作暂存寄存器。如前面所述，通用 SWI 参数和返回值通过 r0～r3 寄存器传递。而 r12 可用于传递通用 SWI 调用的中断功能编号。

下面的例子显示了通用 SWI 的 C 语言程序框架。

```
__swi_indirect(0x80)
unsigned SWI_ManipulateObject(unsigned operationNumber,
unsigned object,unsigned parameter);
unsigned DoSelectedManipulation(unsigned object,
unsigned parameter, unsigned operation)
{
return SWI_ManipulateObject(operation, object, parameter);
}
```

生成的汇编代码如下。

```
DoSelectedManipulation PROC
```

```
    STMFD sp!,{r3,lr}
    MOV r12,r2
    SWI 0x80
    LDMFD sp!,{r3,pc}
    ENDP
```

华清远见

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 4 章 ARM 指令寻址方式

ARM 指令集可以分为跳转指令、数据处理指令、程序状态寄存器传输指令、Load/Store 指令、协处理器指令和异常中断产生指令。根据适用的指令类型不同，指令的寻址方式分为：数据处理指令操作数寻址方式和内存访问指令寻址方式。

4.1 数据处理指令的寻址方式

4.1.1 数据处理指令的寻址方式概要

数据处理指令的基本语法格式如下。

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<shifter_operand>
```

其中<shifter_operand>有下面 11 种形式，如表 4.1 所示。

表 4.1 <shifter_operand>的寻址方式

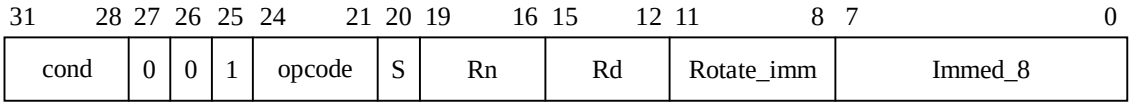
	语 法	寻 址 方 式
1	# <immediate>	立即数寻址
2	<Rm>	寄存器寻址
3	<Rm>, LSL #<shift_imm>	立即数逻辑左移
4	<Rm>, LSL <Rs>	寄存器逻辑左移
5	<Rm>, LSR #<shift_imm>	立即数逻辑右移
6	<Rm>, LSR <Rs>	寄存器逻辑右移
7	<Rm>, ASR #<shift_imm>	立即数算术右移
8	<Rm>, ASR <Rs>	寄存器算术右移
9	<Rm>, ROR #<shift_imm>	立即数循环右移
10	<Rm>, ROR <Rs>	寄存器循环右移
11	<Rm>, RRX	寄存器扩展循环右移

数据处理指令的寻址方式根据<shifter_operand>的不同，相应的分为 11 种。

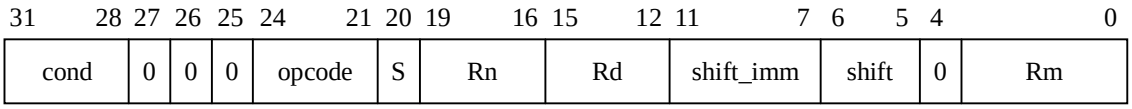
4.1.2 指令解码

图 4.1 显示了数据处理指令不同寻址方式下的解码格式。

32 位立即数



立即数移位



寄存器移位

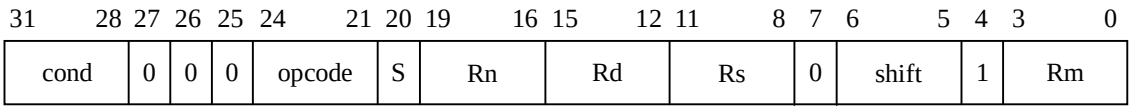


图 4.1　数据操作指令编码格式

编码格式中各域含义如下。

- <opcode>：确定具体指令。
- S：标识指令是否影响程序状态寄存器 CPSR 条件标志。
- Rd：指令操作的目的寄存器。
- Rn：指令第一源操作数。
- bit[11：0]：移位操作，详见本章移位操作一节。
- bit[25]：被用来区分是立即数移位操作还是寄存器移位操作。

如果指令编码出现下面情况：bit[25] = 0 并且 bit[4] = 1 并且 bit[7] = 1，则指令并非数据处理指令，它可能是 Load/Store 指令或算术指令。

4.1.3　移位操作

数据处理指令是在算术逻辑单元 ALU 中完成。ARM 处理器一个显著特征就是可以在操作数进入 ALU 之前，对操作数进行指定位数的左移或右移操作。这种功能明显增强了数据处理操作的灵活性。

移位操作可能产生进位，更新程序状态寄存器 CPSR 的进位标志 C。移位操作有下面 3 种基本方式。

1．立即数方式

没有任何一条 ARM 指令可以包含一个 32 位的立即数，数据处理指令编码格式中，第二个操作数有 12 位。指令的编码格式如图 4.1 所示。

指令中的立即数是由一个 8 bit 的常数移动 4 bit 偶数位（0，2，4，…，26，28，30）得到的。所以，每一条指令都包含一个 8 bit 的常数 X 和移位值 Y，得到的立即数=X 循环

右移（2×Y）。

注意 8 位立即数一定要移偶数位。

下面列举了一些有效的立即数。

0xFF、0x104、0xFF0、0x FF00、0x FF000、0x FF000000、0x F000000F

下面是一些无效的立即数。

0x101、0x102、0x FF1、0x FF04、0x FF003、0x FFFFFFFF、0x F000001F

下面是一些应用立即数的指令。

```
MOV  r0，＃0                ;送 0 到 r0
ADD  r3，r3，＃1            ;r3 的值加 1
CMP  r7，＃1000             ;r7 的值和 1000 比较
BIC  r9，r8，＃0x FF00      ;将 r8 中 8～15 位清零，结果保存在 r9 中
```

2．寄存器方式

寄存器的值可以被直接用于数据操作指令，如：

```
MOV  r2，r0                 ;r0 的值送 r2
ADD  r4，r3，r2             ;r2 加 r3，结果送 r4
CMP  r7，r8                 ;比较 r7 和 r8 的值
```

3．寄存器移位方式

寄存器的值在被送到 ALU 之前，可以事先经过桶形移位寄存器的处理。预处理和移位发生在同一周期内，所以有效的使用移位寄存器，可以增加代码的执行效率。

具体的移位（或者循环移位）方式有下面几种。

- ASR：算术右移。
- LSL：逻辑左移。
- LSR：逻辑右移。
- ROR：循环右移。
- RRX：扩展的循环右移。

以上 5 种移位方式，移位值均可以由立即数或寄存器指定。下面是一些在指令中使用了移位操作的例子。

```
ADD  r2,r0,r1,LSR  #5
MOV  r1,r0,LSL  #2
RSB  r9,r5,r5,LSL  #1
SUB  r1,r2,r0,LSR ＃4
MOV  r2,r4,ROR  r0
```

4.1.4 寻址方式分类详解

数据处理指令的寻址方式根据<shifter_operand>的不同，相应的分为 11 种。详见表 4.1。下面对各类寻址方式进行详细说明。

1．＃<immediate>

（1）编码格式

指令的编码格式如图 4.2 所示。

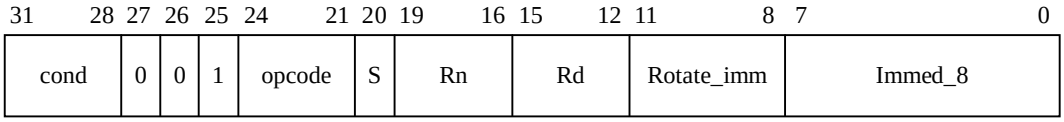


图 4.2 数据处理指令——立即数寻址编码格式

立即数寻址为数据处理指令提供了一个可直接操作的立即数。立即数的生成方法见前面章节介绍。如果移位值为 0，则移位进位值为程序状态寄存器 CPSR 的 C 标志位；否则，为 32-bit 立即数的 bit[31]。

（2）操作伪代码

```
Shifter_operand = immed_8 Rotate_Right (rotate_imm*2)
if  rotate_imm == 0 then
     shifter_carry_out = C flag
else  /* rotate_imm != 0*/
     shifter_carry_out = shifter_operand[31]
```

（3）说明

① 并不是所有的 32-bit 立即数都是可以使用的合法立即数。只有那些通过将一个 8-bit 的立即数循环右移偶数位可以得到的立即数才可以在指令中使用。

② 有些立即数可以通过不止一种方法得到。由于立即数的构造方法中移位包含了循环操作，而循环移位操作会影响 CPSR 的条件标志位 C。因此，同一个合法的立即数由于采用了不同的编码方式，将使这些指令的执行产生不同的结果，这是不能允许的。ARM 汇编器按照下面的规则来生成立即数的编码。

- 当立即数数值在 0 和 0xFF 范围时，令 immed_8=<immediate>，immed_4=0。
- 其他情况下，汇编编译器选择使用 immed_4 数值最小的编码方式。

③ 为了更精确地控制立即数的生成，可以使用下面的语法格式控制立即数的生成。

```
#<immed_8>,<rotate_amout>
```

其中，<rotate_amout> = 2*rotate_imm

（4）举例

```
SUBS  r0,r0,#1                    ;寄存器 r0 中的数值减 1，结果保存到 r0
```

```
MOV  r0,#0xff00     ; 0xff00 → r0;将立即数 0xff00 放入 r0 保存
```

2．<Rm>

（1）编码格式

指令的编码格式如图 4.3 所示。

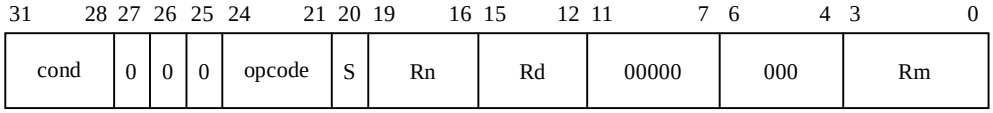


图 4.3　数据处理指令——寄存器寻址编码格式

指令的操作数即为寄存器中的数值。移位寄存器的进位为程序状态寄存器 CPSR 的 C 标志位。

指令的语法格式为：<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>

（2）操作伪代码

```
Shifter_operand = Rm
Shifter_carry_out = C Flag
```

（3）说明

① 从指令的解码格式来看，寄存器寻址方式和使用立即数逻辑左移寻址解码格式是相同的，只是其移位数为 0。

② 如果指令中的 Rm 或 Rn 指定为程序计数器 r15，则操作数的值为当前指令地址加 8。

（4）举例

```
MOV  r1,r2    ; r2 → r1
SUB  r0,r1,r2   ; r1 – r2 → r0
```

3．<Rm>, LSL　#<shift_imm>

（1）编码格式

指令的编码格式如图 4.4 所示。

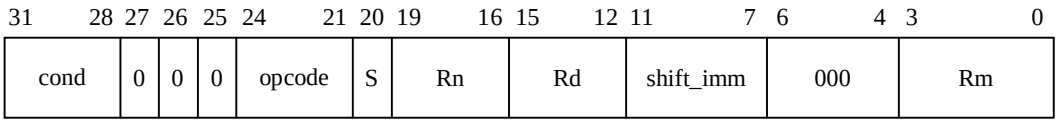


图 4.4　数据处理指令——立即数逻辑左移寻址编码格式

指令的操作数为寄存器 Rm 的数值逻辑左移 shift_imm 位。左移的范围在 0 到 31 之间。左移移出的位用 0 补齐。进位标志位是最后移出的位（如果移位数为 0，则为 C 标志位）。

指令的语法格式为：<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>,LSL #<shift_imm>，其中：

- <Rm>为进行逻辑左移操作的寄存器；
- LSL 为逻辑左移操作标识；

- <shift_imm>为逻辑左移位数，范围为 0～31。

（2）操作伪代码

```
if  shift_imm == 0 then /*执行寄存器操作*/
     shifter_operand = Rm
     shifter_carry_out = C flag
else  /*移位寄存器大于零*/
     shifter_operand = Rm logical_shift_left shift_imm
     shifter_carry_out = Rm[32 – shift_imm]
```

（3）说明

① 如果移位立即数<shift_imm> =0，则该寻址方式为立即数直接寻址。

② 如果指令中的 Rm 或 Rn 指定为程序计数器 r15，则操作数的值为当前指令地址加 8。

（4）举例

```
SUB r0，r1，r2，LSL #10     ;r1 的值减去 r2 的值左移 10bit，结果放到 r0 寄存器
MOV r0，r2，LSL #3          ;r2 的值左移 3bit，结果放入 r0，即 r0 = r2×8
```

4．<Rm>, LSL　<Rs>

（1）编码格式

指令的编码格式如图 4.5 所示。

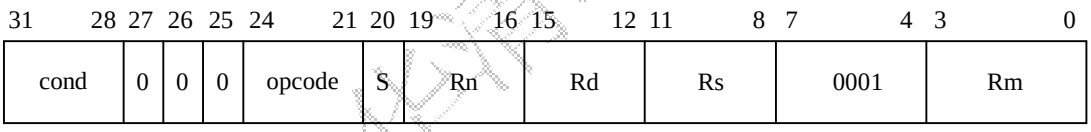


图 4.5　数据处理指令——寄存器逻辑左移寻址编码格式

寄存器逻辑左移十分适合寄存器值乘 2 的倍数操作。

这个指令是将寄存器 Rm 的值逻辑左移一定的位数。位移的位数由 Rs 的最低 8 位 bit[7：0]决定。Rm 移出的位用 0 补齐。进位值是移位寄存器最后移出的位，如果移位数大于 0，则进位值为 0。

（2）语法格式

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>,LSL  <Rs>
```

其中：

- <Rm>为指令被移位的寄存器；
- LSL 为逻辑左移操作标识；
- <Rs>为包含逻辑左移位数的寄存器。

（3）操作伪代码

```
if  Rs[7:0] = = 0 then
```

```
        shifter_operand = Rm
        shifter_carry_out = C flag
    else  if  Rs[7:0] < 32 then
        shifter_operand = Rm  logical_shift_left  Rs[7:0]
        shifter_carry_out = Rm[32 – Rs[7:0]]
    else  if  Rs[7:0] = = 32 then
        shifter_operand = 0
        shifter_carry_out = Rm[0]
    else  /*Rs 的后 8 位大于零*/
        shifter_operand = 0
        shifter_carry_out = 0
```

（4）说明

如果程序计数器 r15 被用作 Rd，Rm，Rn 或 Rs 中的任意一个，则指令的执行结果不可预知。

（5）举例

```
MOV  r0，r2，LSL r3       ;r2 的值左移 r3 位，结果放入 r0
ANDS r1，r1，r2，LSL r3   ;r2 的值左移 r3 位，然后和 r1 相与，结果放入 r1
```

5．<Rm>, LSR ＃<shift_imm>

（1）编码格式

指令的编码格式如图 4.6 所示。

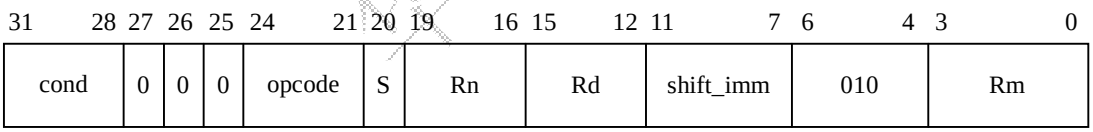


图 4.6　数据处理指令——立即数逻辑右移寻址编码格式

指令的操作数为寄存器 Rm 的值右移<shift_imm>位，相当于 Rm 的值除以一个 2 的倍数。<shift_imm>值的范围为 0～31，移位后空出的位添 0。循环器进位值为 Rm 最后移出的位。

（2）语法格式

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>,LSR ＃<shift_imm>
```

其中：

- <Rm>为被移位的寄存器；
- LSR 为逻辑右移操作标识；
- <shift_imm>为逻辑右移位数，范围为 0～31。

（3）操作伪代码

```
if  shift_imm == 0 then /*执行寄存器操作*/
     shifter_operand = 0
     shifter_carry_out = Rm[31]
else  /*移位立即数大于零*/
     shifter_operand = Rm logical_shift_Right shift_imm
     shifter_carry_out = Rm[shift_imm - 1]
```

（4）说明

① shift_imm 的取值范围为 0～31，当 shift_imm=0 时，移位位数为 32，所以移位位数范围为 1～32 位。

② 如果指令中的 Rm 或 Rn 指定为程序计数器 r15，则操作数的值为当前指令地址加 8。

6．<Rm>, LSR　<Rs>

（1）编码格式

指令的编码格式如图 4.7 所示。

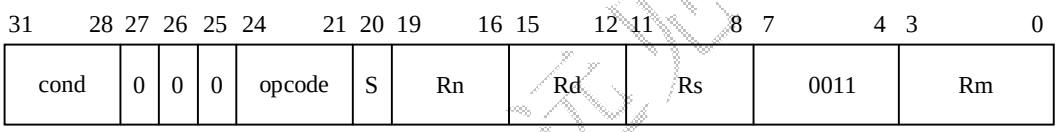


图 4.7　数据处理指令——寄存器逻辑右移寻址编码格式

此操作将寄存器 Rm 的数值逻辑右移一定的位数。移位的位数由 Rs 的最低 8 位 bit[7：0]决定。移出的位由 0 补齐。当 Rs[7：0]大于 0 而小于 32 时，进位标志 C 由最后移出的位决定，当 Rs[7：0]大于 32 时，进位标志位为 0，当 Rs[7：0]等于 0 时，进位标志不变。

（2）语法格式

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>,LSR  <Rs>
```

其中：

- <Rm>为指令被移位的寄存器；
- LSR 为逻辑右移操作标识；
- <Rs>为包含逻辑右移位数的寄存器。

（3）操作伪代码

```
if  Rs[7:0] = = 0 then
     shifter_operand = Rm
     shifter_carry_out = C flag
else  if  Rs[7:0] < 32 then
     shifter_operand = Rm  logical_shift_Right  Rs[7:0]
     shifter_carry_out = Rm[Rs[7:0] - 1]
else  if  Rs[7:0] = = 32 then
```

```
        shifter_operand = 0
        shifter_carry_out = Rm[31]
    else  /*Rs 的后 8 位大于零*/
        shifter_operand = 0
        shifter_carry_out = 0
```

（4）说明

如果程序计数器 r15 被用作 Rd、Rm、Rn 或 Rs 中的任意一个，则指令的执行结果不可预知。

7．<Rm>, ASR ＃<shift_imm>

（1）编码格式

指令的编码格式如图 4.8 所示。

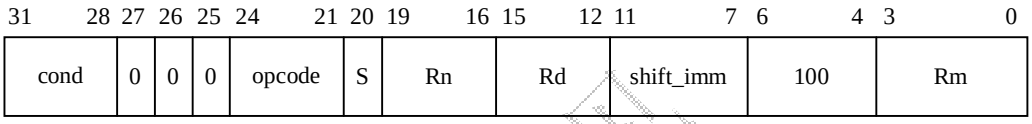


图 4.8　数据处理指令——立即数算术右移寻址编码格式

指令的操作数为寄存器 Rm 的数值逻辑右移<shift_imm>位。<shift_imm>的值范围为 0～31，当<shift_imm>等于 0 时，移位位数为 32，所以移位位数范围为 1～32 位。进位移位操作后，空出的位添 Rm 的最高位 Rm[31]。进位标志为 Rm 最后被移出的数值。

（2）语法格式

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>,ASR ＃<shift_imm>
```

其中：

- <Rm>为被移位的寄存器；
- ASR 为算术右移操作标识；
- <shift_imm>为算术右移位数，范围为 1～32，当 shift_imm 等于 0 时移位位数为 32。

（3）操作伪代码

```
if  shift_imm == 0 then /*执行寄存器操作*/
    if  Rm[31] = = 0 then
        shifter_operand = 0
        shifter_carry_out = Rm[31]
    else /*Rm[31] = = 1*/
        shifter_operand = 0xffffffff
        shifter_carry_out = Rm[31]
else  /*shift_imm > 0*/
    shifter_operand = Rm Arithmetic_shift_Right <shift_imm>
```

```
    shifter_carry_out = Rm[shift_imm - 1]
```

（4）说明

① 如果指令中的 Rm 或 Rn 指定为程序计数器 r15，则操作数的值为当前指令地址加 8。

8．<Rm>, ASR　<Rs>

（1）编码格式

指令的编码格式如图 4.9 所示。

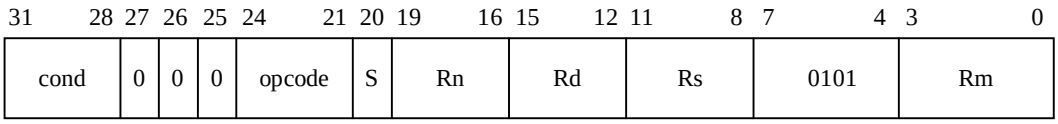


图 4.9　数据处理指令——寄存器算术右移寻址编码格式

此操作将寄存器 Rm 的数值算术右移一定的位数。移位后空缺的位由 Rm 的符号位（Rm[31]）填充。位移的位数由 Rs 的最低 8 位 bit[7：0]决定。当 Rs[7：0]大于零而小于 32 时，指令的操作数为寄存器 Rm 的数值算术右移 Rs[7：0]位，进位标志 C 为 Rm 最后被移出的位。

（2）语法格式

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>,ASR  <Rs>
```

其中：

- <Rm>为指令被移位的寄存器；
- ASR 为算术右移操作标识；
- <Rs>为包含算术右移位数的寄存器。

（3）操作伪代码

```
if  Rs[7:0] = = 0 then
    shifter_operand = Rm
    shifter_carry_out = C flag
else  if  Rs[7:0] < 32 then
    shifter_operand = Rm  Arithmeticl_shift_Right  Rs[7:0]
    shifter_carry_out = Rm[Rs[7:0] - 1]
else
    if  Rm[31] = =0 then
        shifter_operand = 0
        shifter_carry_out = Rm[31]
   else
        shifter_operand = 0xffffffff
        shifter_carry_out = Rm[31]
```

（4）说明

如果程序计数器 r15 被用作 Rd、Rm、Rn 或 Rs 中的任意一个，则指令的执行结果不可

预知。

9．<Rm>, ROR ＃<shift_imm>

（1）编码格式

指令的编码格式如图 4.10 所示。

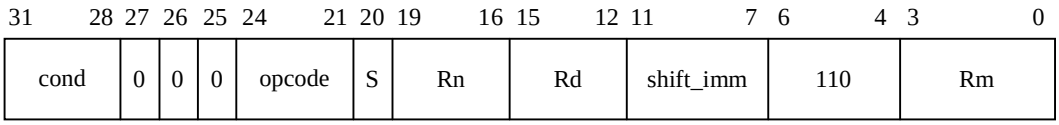


图 4.10　数据处理指令——立即数循环右移寻址编码格式

指令的操作数由寄存器 Rm 的数值循环右移一定的位数得到。移位的位数由 Rs 的最低 8 位 bits[7：0]决定。当 Rs[7：0]=0 时，指令的操作数为寄存器 Rm 的值，循环器的进位值为 CPSR 中的 C 条件标志位；否则，循环器的进位值为 Rm 最后被移出的位。

（2）语法格式

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>,ROR ＃<shift_imm>
```

其中：

- <Rm>为被移位的寄存器；
- ROR 为循环右移操作标识；
- <shift_imm>为循环右移位数，范围为 1～31，当 shift_imm 等于 0 时执行 RRX 操作。

（3）操作伪代码

```
if  shift_imm == 0 then /*执行寄存器操作*/
    执行 RRX 操作
else
    shifter_operand = Rm Rotate_Right shift_imm
    shifter_carry_out = Rm[shift_imm - 1]
```

（4）说明

如果指令中的 Rm 或 Rn 指定为程序计数器 r15，则操作数的值为当前指令地址加 8。

10．<Rm>, ROR　<Rs>

（1）编码格式

指令的编码格式如图 4.11 所示。

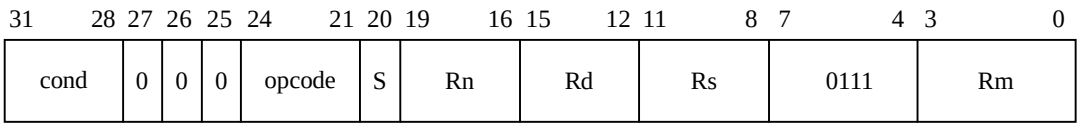


图 4.11　数据处理指令——寄存器循环右移寻址编码格式

指令的操作数由寄存器 Rm 的数值循环右移一定的位数。移位的位数由 Rs 的最低 8 位 bits[7：0]决定。当 Rs[7：0]=0 时，指令的操作数为寄存器 Rm 的值，循环器的进位值为 CPSR 中的 C 条件标志位；否则，循环器的进位值为 Rm 最后被移出的位。

（2）语法格式

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>,ROR  <Rs>
```

其中：

- <Rm>为指令被移位的寄存器；
- ROR 为循环右移操作标识；
- <Rs>为包含循环右移位数的寄存器。

（3）操作伪代码

```
if  Rs[7:0] = = 0 then
    shifter_operand = Rm
    shifter_carry_out = C flag
else  if  Rs[4:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = Rm[31]
else
    shifter_operand = Rm Rotate_Right Rs[4:0]
    shifter_carry_out = Rm[Rs[4:0] - 1]
```

（4）说明

如果程序计数器 r15 被用作 Rd、Rm、Rn 或 Rs 中的任意一个，则指令的执行结果不可预知。

11．<Rm>, RRX

（1）编码格式

指令的编码格式如图 4.12 所示。

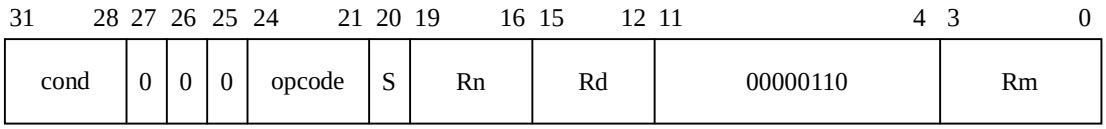


图 4.12　数据处理指令——扩展右移寻址编码格式

指令的操作数为寄存器 Rm 的数值右移一位，并用 CPSR 中的 C 条件标志位填补空出的位。CPSR 中的 C 条件标志位则用移出的位代替。

（2）语法格式

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<Rm>,RRX
```

其中：

- <Rm>为指令被移位的寄存器；
- RRX 为扩展的循环右移操作。

（3）操作伪代码

```
    shifter_operand = (C flag logical_shift_left 31) OR (Rm logical_shift_Right
1)
    shifter_carry_out = Rm[0]
```

（4）说明

① 此种寻址方式的编码形式和“ROR ＃0”一致。

② 如果程序计数器 r15 被用作 Rd、Rm、Rn 或 Rs 中的任意一个，则指令的执行结果不可预知。

③ 可以实现 ADC 指令的功能。

4.2 内存访问指令寻址

根据内存访问指令的分类，内存访问指令的寻址方式可以分为以下几种。

① 字及无符号字节的 Load/Store 指令的寻址方式。

② 杂类 Load/Store 指令的寻址方式。

③ 批量 Load/Store 指令的寻址方式。

④ 协处理器 Load/Store 指令的寻址方式。

4.2.1 字及无符号字节的 Load/Store 指令的寻址方式

字及无符号字节的 Load/Store 指令语法格式如下：

```
    LDR|STR{<cond>}{B}{T}  <Rd>，<addressing_mode>
```

其中<addressing_mode>共有 9 种寻址方式，如表 4.2 所示。

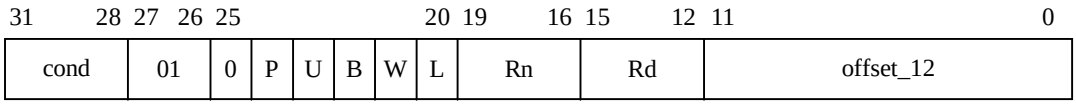
表 4.2 字及无符合字节的 Load/Store 指令的寻址方式

	格 式	模 式
1	[Rn，#±<offset_12>]	立即数偏移寻址 （Immediate offset）
2	[Rn，±Rm]	寄存器偏移寻址 （Register offset）
3	[Rn，Rm，<shift>#< offset_12>]	带移位的寄存器偏移寻址 （Scaled register offset）
4	[Rn，#±< offset_12>]!	立即数前索引寻址 （Immediate pre-indexed）
5	[Rn，±Rm]!	寄存器前索引寻址 （Register post-indexed）
6	[Rn，Rm，<shift>#< offset_12>]!	带移位的寄存器前索引寻址

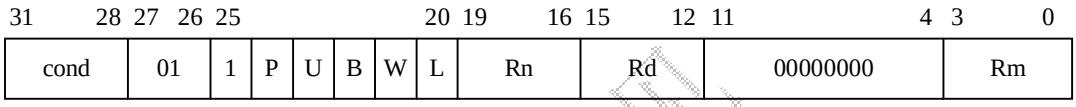
		(Scaled register pre-indexed)
7	[Rn]，#±< offset_12>	立即数后索引寻址 (Immediate post-indeded)
8	[Rn]，±<Rm>	寄存器后索引寻址 (Register post-indexed)
9	[Rn]，±<Rm>，<shift>#< offset_12>	带移位的寄存器后索引寻址 (Scaled register post-indexed)

字及无符号字节的 Load/Store 指令的解码格式如图 4.13 所示。

立即数



寄存数



寄存器偏移

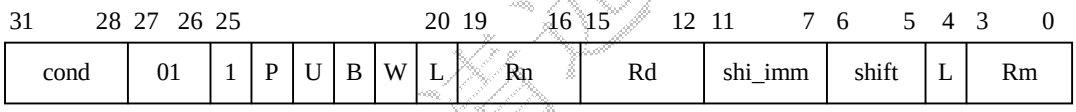


图 4.13　字及无符号字节的 Load/Store 指令的解码格式

编码格式中各位的含义如表 4.3 所示。

表 4.3　　字和无符号半字 Load/Store 指令编码格式各位含义

位　标　识	取　　值	含　　义
P	P=0	使用后索引寻址
	P=1	使用偏移地址或前索引寻址（由 W 位决定）
U	U=0	访问的地址=基址寄存器的值-偏移量（offset）
	U=1	访问的地址=基址寄存器的值+偏移量（offset）
B	B=0	字访问 Load/Store
	B=1	无符号字节访问 Load/Store
W	W=0	如果 P=0，该指令为 LDR、LDRB、STR 或 STRB 指令，且内存访问指令为正常访问指令；如果 P=1，指令执行不更新基地址
	W=1	如果 P=0，该指令为 LDRBT、LDRT、STRBT 或 STRT，且指令为非特权（用户模式）访问指令；如果 P=1，计算内存地址并更新基地址
L	L=0	Store 指令

	L=1	Load 指令

1．[Rn，#±<offset_12>]

（1）编码格式

指令的编码格式如图 4.14 所示。

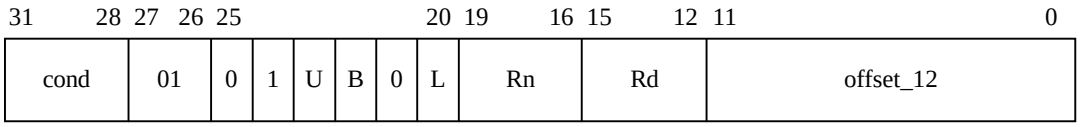


图 4.14　内存访问指令——立即数偏移寻址编码格式

内存访问地址为基址寄存器 Rn 的值加（或减）立即数 offset_12。

编程中，在访问结构体或记录（record）类型的变量时，这些内存的操作指令是十分有效的。另外，在子程序中也常用这些指令访问本地变量和堆栈。

（2）语法格式

```
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，#±<offset_12>]
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <offset_12>为 12 位立即数，内存访问地址偏移量。

（3）操作伪代码

```
If  U = = 1  then
    Address = Rn + offset_12
Else
    Address = Rn – offset_12
```

（4）说明

① 如果指令中没有指定立即数，使用[<Rn>]，编译器按[<Rn>，＃0]形式编码。

② 如果 Rn 被指定为程序计数器 r15，其值为当前指令地址加 8。

2．[Rn，±Rm]

（1）编码格式

指令的编码格式如图 4.15 所示。

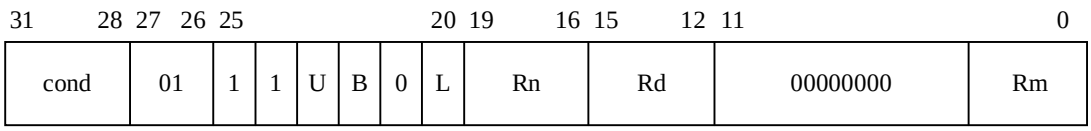


图 4.15　内存访问指令——寄存器偏移寻址编码格式

内存访问地址为基址寄存器 Rn 的值加（或减）偏移寄存器 Rm 的值。

该寻址方式适合使用指针访问字节数组中的数据成员。

（2）语法格式

```
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>]
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <Rm>为偏移地址寄存器，包含内存访问地址偏移量。

（3）操作伪代码

```
If  U = = 1  then
    Address = Rn + Rm
Else
    Address = Rn – Rm
```

（4）说明

如果 Rn 被指定为程序计数器 r15，其值为当前指令地址加 8；如果 r15 被用作偏移地址寄存器 Rm 的值，指令的执行结果不可预知。

3．[Rn，Rm，<shift>#< offset_12>]

（1）编码格式

指令的编码格式如图 4.16 所示。

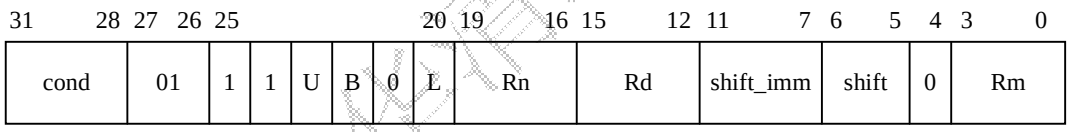


图 4.16　内存访问指令——带移位的寄存器偏移寻址编码格式

内存地址为 Rn 的值加/减通过移位操作后的 Rm 的值。

当数组中的成员长度大于 1 个字节时，使用该寻址方式可高效率地访问数组成员。

（2）语法格式

语法格式有以下 5 种。

```
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，LSL #< offset_12>]
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，LSR #< offset_12>]
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，ASR #< offset_12>]
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，ROR #< offset_12>]
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，RRX]
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <Rm>为偏移地址寄存器，包含内存访问地址偏移量；
- LSL 表示逻辑左移操作；

- LSR 表示逻辑右移操作；
- ASR 表示算术右移操作；
- ROR 表示循环右移操作；
- RRX 表示扩展的循环右移。
- <shift_imm>为移位立即数。

（3）操作伪代码

```
Case  shift  of
    0b00  /*LSL*/
         Index = Rm logic_shift_left  shift_imm
    0b01  /*LSR*/
         If  shift_imm = = 0  then /*LSR  #32*/
             Index = 0
         Else
             Index = Rm logical_shift_right  shift_imm
    0b10  /*ASR*/
         If  shift_imm = = 0 then /*ASR  #32*/
             If  Rm[31] = = 1 then
                 Index = 0xffffffff
             Else
                 Index = 0
         Else
             Index = Rm  Arithmetic_shift_Right  shift_imm
    0b11  /* ROR or RRX*/
         If  shift_imm = = 0 then /*RRX*/
             Index = (C flag Logical_shift_left 31) OR
                     (Rm logical_shift_Right 1)
         Else /*ROR*/
             Index = Rm Rotate_Right shift_imm
Endcase
If  U = = 1 then
    Address = Rn + index
Else /*U = = 0*/
    Address = Rn – index
```

（4）说明

如果 Rn 被指定为程序计数器 r15，其值为当前指令地址加 8；如果 r15 被用作偏移地址寄存器 Rm 的值，指令的执行结果不可预知。

4．[Rn，#±< offset_12>]!

（1）编码格式

指令的编码格式如图 4.17 所示。

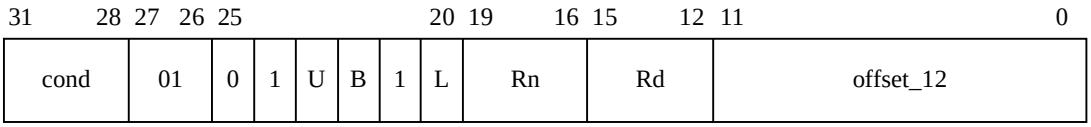


图 4.17　内存访问指令——前索引立即数偏移寻址编码格式

内存地址为基址寄存器 Rn 加/减立即数 offset_8 的值。当指令执行的条件<cc>满足时，生成的地址写回基址寄存器 Rn 中。

该寻址方式适合访问数组自动进行数组下标的更新。

（2）语法格式

```
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<offset_12>]！
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <offset_12>为 12 位立即数，内存访问地址偏移量；
- ！设置指令编码中的 W 位，更新指令基址寄存器。

（3）操作伪代码

```
If  U == 1  then
    Address = Rn + offset_12
Else
    Address = Rn – offset_12
If  ConditionPassed{cond}  then
    Rn = address
```

（4）说明

① 如果指令中没有指定立即数，使用[<Rn>]，编译器按[<Rn>，＃0]！形式编码。

② 如果 Rn 被指定为程序计数器 r15，指令的执行结果不可预知。

5．[Rn，±Rm]!

（1）编码格式

指令的编码格式如图 4.18 所示。

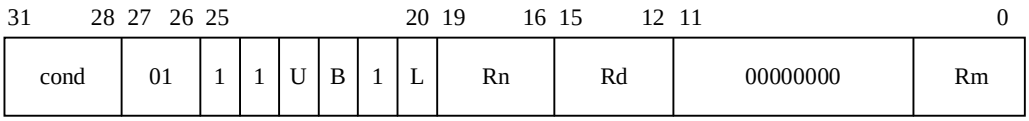


图 4.18　内存访问指令——前索引寄存器偏移寻址编码格式

内存访问地址为基址寄存器 Rn 的值加（或减）偏移寄存器 Rm 的值。当指令的执行条件<cc>满足时，生成地地址将写回基址寄存器。

（2）语法格式

```
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>]
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <Rm>为偏移地址寄存器，包含内存访问地址偏移量；
- ！设置指令编码中的 W 位，更新指令基址寄存器。

（3）操作伪代码

```
If  U = = 1  then
    Address = Rn + Rm
Else
    Address = Rn – Rm
If  ConditionPassed{cond}  then
    Rn = address
```

（4）说明

如果 Rn 和 Rm 指定为同一寄存器，指令的执行结果不可预知。

6．[Rn，±Rm，<shift>#< offset_12>]！

（1）编码格式

指令的编码格式如图 4.19 所示。

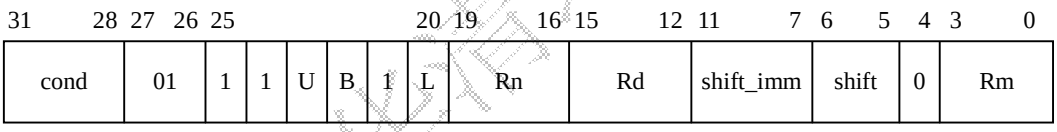


图 4.19　内存访问指令——带移位的前索引寄存器偏移寻址编码格式

内存地址为 Rn 的值加/减通过移位操作后的 Rm 的值。当指令的执行条件<cc>满足时，生成地地址将写回基址寄存器。

（2）语法格式

语法格式有以下 5 种。

```
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，LSL #< offset_12>] !
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，LSR #< offset_12>] !
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，ASR #< offset_12>] !
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，ROR #< offset_12>] !
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>，±<Rm>，RRX] !
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <Rm>为偏移地址寄存器，包含内存访问地址偏移量；
- LSL 表示逻辑左移操作；
- LSR 表示逻辑右移操作；

- ASR 表示算术右移操作；
- ROR 表示循环右移操作；
- RRX 表示扩展的循环右移。
- <shift_imm>为移位立即数。
- ！设置指令编码中的 W 位，更新指令基址寄存器。

（3）操作伪代码

```
Case  shift  of
    0b00  /*LSL*/
         Index = Rm logic_shift_left  shift_imm
    0b01  /*LSR*/
         If  shift_imm = = 0  then /*LSR  #32*/
             Index = 0
         Else
             Index = Rm logical_shift_right  shift_imm
    0b10  /*ASR*/
         If  shift_imm = = 0 then /*ASR  #32*/
             If  Rm[31] = = 1 then
                 Index = 0xffffffff
             Else
                 Index = 0
         Else
             Index = Rm  Arithmetic_shift_Right  shift_imm
    0b11  /* ROR or RRX*/
         If  shift_imm = = 0 then /*RRX*/
             Index = (C flag Logical_shift_left 31) OR
                     (Rm logical_shift_Right 1)
         Else /*ROR*/
             Index = Rm Rotate_Right shift_imm
Endcase
If  U = = 1 then
    Address = Rn + index
Else /*U = = 0*/
    Address = Rn – index
If  ConditionPassed{cond}  then
    Rn = address
```

（4）说明

① 当 PC 用作基址寄存器 Rn 或 Rm 时，指令执行结果不可预知。

② 当 Rn 和 Rm 是同一个寄存器时，指令的执行结果不可预知。

7．[Rn]，#±< offset_12>

（1）编码格式

指令的编码格式如图 4.20 所示。

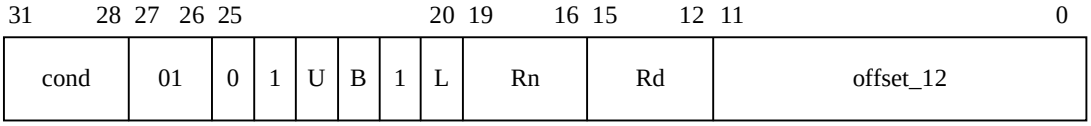


图 4.20 内存访问指令——后索引立即数偏移寻址编码格式

指令使用基址寄存器 Rn 的值作为实际内存访问地址。当指令的执行条件满足时，将基址寄存器的值加/减偏移量产生新的地址值回写到 Rn 寄存器中。

（2）语法格式

```
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>]，±<offset_12>
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <offset_12>为 12 位立即数，内存访问地址偏移量。

（3）操作伪代码

```
Address = Rn
If  conditionPassed{cond}  then
    If  U = = 1 then
        Rn = Rn + offset_12
    Else
        Rn = Rn – offset_12
```

（4）说明

① LDRBT、LDRT、STRBT 和 STRT 指令只支持后索引寻址。

② 如果 Rn 被指定为程序计数器 r15，指令的执行结果不可预知。

8．[Rn]，±<Rm>

（1）编码格式

指令的编码格式如图 4.21 所示。

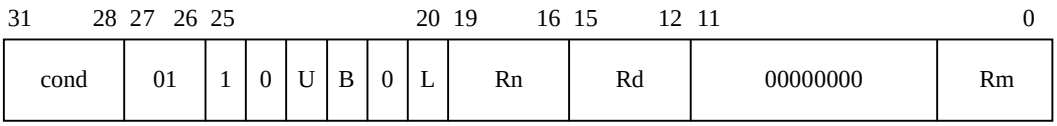


图 4.21 内存访问指令——后索引寄存器偏移寻址编码格式

指令访问地址为实际的基址寄存器的值。当指令的执行条件满足时，将基址寄存器的值加/减索引寄存器 Rm 的值回写到 Rn 基址寄存器。

（2）语法格式

```
LDR|STR{<cond>}{B}{T}  <Rd>，[Rn]，±<Rm>
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <Rm>为偏移地址寄存器，包含内存访问地址偏移量。

（3）操作伪代码

```
Address = Rn
If  conditionPassed{cond}  then
    If  U = = 1  then
        Rn = Rn + Rm
    Else
        Rn = Rn – Rm
```

（4）说明

① LDRBT、LDRT、STRBT 和 STRT 指令只支持后索引寻址。

② 如果 Rm 和 Rn 指定为同一寄存器，指令的执行结果不可预知。

9．[Rn]，±Rm，<shift>#< offset_12>]

（1）编码格式

指令的编码格式如图 4.22 所示。

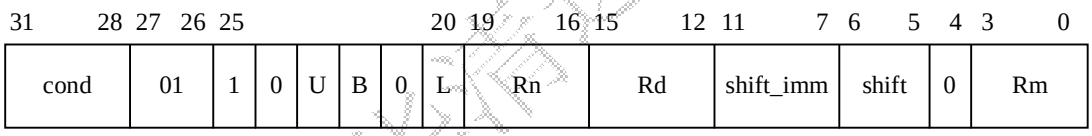


图 4.22　内存访问指令——带移位的后索引寄存器偏移寻址编码格式

实际的内存访问地址为寄存器 Rn 的值。当指令的执行条件满足时，将基址寄存器值加/减一个地址偏移量产生新的地址值。

（2）语法格式

语法格式有以下 5 种。

```
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>]，±<Rm>，LSL #< offset_12>
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>]，±<Rm>，LSR #< offset_12>
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>]，±<Rm>，ASR #< offset_12>
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>]，±<Rm>，ROR #< offset_12>
LDR|STR{<cond>}{B}{T}  <Rd>，[<Rn>]，±<Rm>，RRX
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <Rm>为偏移地址寄存器，包含内存访问地址偏移量；
- LSL 表示逻辑左移操作；
- LSR 表示逻辑右移操作；

- ASR 表示算术右移操作；
- ROR 表示循环右移操作；
- RRX 表示扩展的循环右移。
- <shift_imm>为移位立即数。

（3）操作伪代码

```
Address = Rn
Case  shift  of
    0b00  /*LSL*/
         Index = Rm logic_shift_left  shift_imm
    0b01  /*LSR*/
         If  shift_imm = = 0  then /*LSR  #32*/
             Index = 0
         Else
             Index = Rm logical_shift_right  shift_imm
    0b10  /*ASR*/
         If  shift_imm = = 0 then /*ASR  #32*/
             If  Rm[31] = = 1 then
                 Index = 0xffffffff
             Else
                 Index = 0
         Else
             Index = Rm  Arithmetic_shift_Right  shift_imm
    0b11  /* ROR or RRX*/
         If  shift_imm = = 0 then /*RRX*/
             Index = (C flag Logical_shift_left 31) OR
                     (Rm logical_shift_Right 1)
         Else /*ROR*/
             Index = Rm Rotate_Right shift_imm
Endcase
If  ConditionPassed{cond}  then
   If  U = = 1 then
       Rn = Rn + index
   Else /*U = = 0*/
       Rn = Rn – index
```

（4）说明

① LDRBT、LDRT、STRBT 和 STRT 指令只支持后索引寻址。

② 当 PC 用作基址寄存器 Rn 或 Rm 时，指令执行结果不可预知。

③ 当 Rn 和 Rm 是同一个寄存器时，指令的执行结果不可预知。

4.2.2 杂类 Load/Store 指令的寻址方式

使用该类寻址方式的指令的语法格式如下。

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>，<addressing_mode>
```

使用该类寻址方式的指令包括：（有符号/无符号）半字 Load/Store 指令、有符号字节 Load/Store 指令和双字 Load/Store 指令。

该类寻址方式分为 6 种类型，如表 4.4 所示。

表 4.4　　杂类 Load/Store 指令的寻址方式

	格　　式	模　　式
1	[Rn，#±<offset_8>]	立即数偏移寻址 （Immediate offset）
2	[Rn，±Rm]	寄存器偏移寻址 （Register offset）
3	[Rn，#±< offset_8>]!	立即数前索引寻址 （Immediate pre-indexed）
4	[Rn，±Rm]!	寄存器前索引寻址 （Register post-indexed）
5	[Rn]，#±< offset_8>	立即数后索引寻址 (Immediate post-indeded)
6	[Rn]，±<Rm>	寄存器后索引寻址 (Register post-indexed)

杂类 Load/Store 指令的解码格式如图 4.23 所示。

立即数

31 28	27 26 25	24				20	19 16	15 12	11 8	7	6	5	4	3 0
cond	000	P	U	1	W	L	Rn	Rd	immedH	1	S	H	1	immedL

寄存数

31 28	27 26 25					20	19 16	15 12	11 8	7	6	5	4	3 0
cond	000	P	U	0	W	L	Rn	Rd	SBZ	1	S	H	1	Rm

图 4.23　杂类 Load/Store 指令解码格式

编码格式中各标志位的含义如表 4.5 所示。

表 4.5　　杂类 Load/Store 指令编码格式各标志位含义

位 标 识	取　　值	含　　义

P	P=0	使用后索引寻址
	P=1	使用偏移地址或前索引寻址（由 W 位决定）

续表

位 标 识	取 值	含 义
U	U=0	访问的地址=基址寄存器的值-偏移量（offset）
	U=1	访问的地址=基址寄存器的值+偏移量（offset）
W	W=0	如果 P=0，使用后索引寻址；P=1，指令不改变基址寄存器的值
	W=1	如果 P=0，未定义指令；如果 P=1，将计算的内存访问地址回写到基址寄存器
L	L=0	Store 指令
	L=1	Load 指令
S	S=0	无符号半字内存访问
	S=1	有符号半字内存访问
H	H=0	字节访问
	H=1	半字访问

注意

当 S=0 并且 H=0 时，并非无符号的字节内存访问指令。无符号的内存访问指令不使用该种寻址方式，详见本章上一节。

当 S=1 并且 L=0 时，并非是有符号的存储指令，而是未定义指令。ARM 指令并未区分有符号和无符号的字节和半字存储。

1．[Rn，#±<offset_8>]

（1）编码格式

指令的编码格式如图 4.24 所示。

立即数

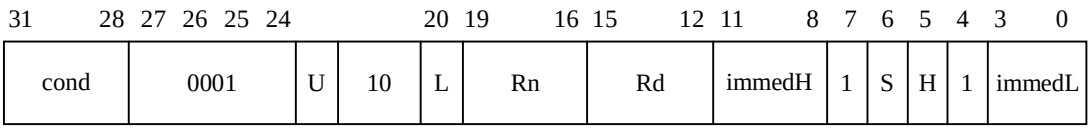


图 4.24 杂项内存访问指令——立即数偏移寻址编码格式

内存访问地址为基址寄存器 Rn 的值加（或减）立即数 offset_8。

编程中，在访问结构体或记录（record）类型的变量时，这些内存的操作指令是十分有效的。另外，在子程序中，也常用这些指令访问本地变量和堆栈。当 offset_8=0 时，内存访问地址即基址寄存器 Rn 的值。

（2）语法格式

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>，[<Rn>，#±<offset_12>]
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址。
- <offset_8>为 8 位立即数，内存访问地址偏移量。

（3）操作伪代码

```
offset_8 = (immedH << 4) OR immedL
If  U = = 1  then
    Address = Rn + offset_8
Else
    Address = Rn – offset_8
```

（4）说明

① 如果指令中没有指定立即数，使用[<Rn>]，编译器按[<Rn>，＃0]形式编码。

② 如果 Rn 被指定为程序计数器 r15，其值为当前指令地址加 8。

2．[Rn，±Rm]

（1）编码格式

指令的编码格式如图 4.25 所示。

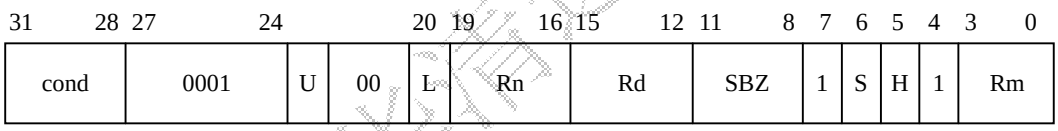


图 4.25　杂项内存访问指令——寄存器偏移寻址编码格式

内存访问地址为基址寄存器 Rn 的值加（或减）偏移寄存器 Rm 的值。

该寻址方式适合使用指针访问数组中的单个数据成员。

（2）语法格式

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>，[<Rn>，±<Rm>]
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <Rm>为偏移地址寄存器，包含内存访问地址偏移量。

（3）操作伪代码

```
If  U = = 1  then
    Address = Rn + Rm
Else
    Address = Rn – Rm
```

（4）说明

如果 Rn 被指定为程序计数器 r15，其值为当前指令地址加 8；如果 r15 被用作偏移地址寄存器 Rm 的值，指令的执行结果不可预知。

3．[Rn，#±< offset_8>]!

（1）编码格式

指令的编码格式如图 4.26 所示。

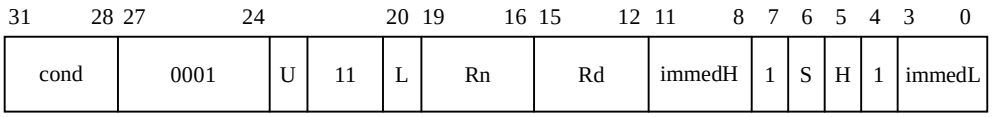


图 4.26　杂类内存访问指令——前索引立即数偏移寻址编码格式

内存地址为基址寄存器 Rn 加/减立即数 offset_8 的值。当指令执行的条件<cc>满足时，生成的地址写回基址寄存器 Rn 中。

该寻址方式适合访问数组自动进行数组下标的更新。

（2）语法格式

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>，[<Rn>，±<offset_8>] ！
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <offset_8>为 8 位立即数，内存访问地址偏移量，在指令编码格式中被拆为 immedH 和 immedL 两部分；
- ！设置指令编码中的 W 位，更新指令基址寄存器。

（3）操作伪代码

```
offset_8 = (immedH) << 4 OR immedL
If  U == 1  then
    Address = Rn + offset_8
Else
    Address = Rn – offset_8
If  ConditionPassed{cond}  then
    Rn = address
```

（4）说明

① 如果指令中没有指定立即数，使用[<Rn>]，编译器按[<Rn>，＃0]！形式编码。

② 如果 Rn 被指定为程序计数器 r15，指令的执行结果不可预知。

4．[Rn，±Rm] ！

（1）编码格式

指令的编码格式如图 4.27 所示。

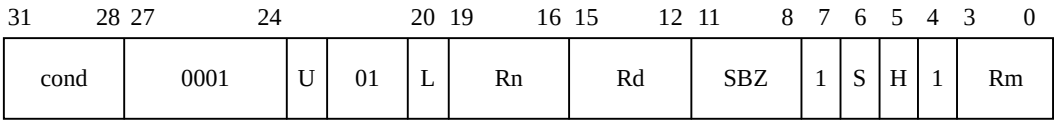


图 4.27　杂项内存访问指令——前索引寄存器偏移寻址编码格式

内存访问地址为基址寄存器 Rn 的值加（或减）偏移寄存器 Rm 的值。当指令的执行条件<cc>满足时，生成地地址将写回基址寄存器。

（2）语法格式

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>，[<Rn>，±<Rm>]
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <Rm>为偏移地址寄存器，包含内存访问地址偏移量；
- ！设置指令编码中的 W 位，更新指令基址寄存器。

（3）操作伪代码

```
If  U = = 1  then
    Address = Rn + Rm
Else
    Address = Rn – Rm
If  ConditionPassed{cond}  then
    Rn = address
```

（4）说明

① 如果 Rn 和 Rm 指定为同一寄存器，指令的执行结果不可预知。

② 如果程序计数器 r15 被用作 Rm 或 Rn，则指令的执行结果不可预知。

5．[Rn]，#±< offset_8>

（1）编码格式

指令的编码格式如图 4.28 所示。

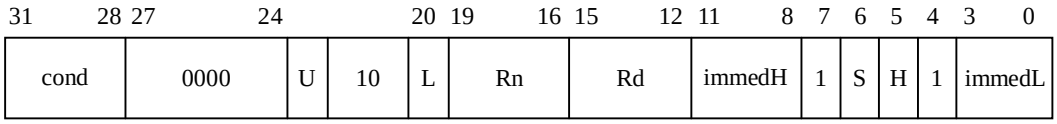


图 4.28　杂项内存访问指令——后索引立即数偏移寻址编码格式

指令使用基址寄存器 Rn 的值作为实际内存访问地址。当指令的执行条件满足时，将基址寄存器的值加/减偏移量生产新的地址值回写到 Rn 寄存器中。

（2）语法格式

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>，[<Rn>]，±<offset_8>
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <offset_8>为 8 位立即数，内存访问地址偏移量。

（3）操作伪代码

```
Address = Rn
Offset_8 = (immedH << 4) OR immedL
If  conditionPassed{cond}  then
    If  U = = 1 then
        Rn = Rn + offset_8
    Else
        Rn = Rn – offset_8
```

（4）说明

① 当指令中没有指定立即数时，汇编器按“[<Rn>]，＃0”编码。

② 如果 Rn 被指定为程序计数器 r15，指令的执行结果不可预知。

6．[Rn]，±<Rm>

（1）编码格式

指令的编码格式如图 4.29 所示。

31–28	27–24	23	22–21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0000	U	00	L	Rn	Rd	SBZ	1	S	H	1	Rm

图 4.29　杂项内存访问指令——后索引寄存器偏移寻址编码格式

指令访问地址为实际的基址寄存器的值。当指令的执行条件满足时，将基址寄存器的值加/减索引寄存器 Rm 的值回写到 Rn 基址寄存器。

（2）语法格式

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>，[Rn]，±<Rm>
```

其中：

- Rn 为基址寄存器，该寄存器包含内存访问的基地址；
- <Rm>为偏移地址寄存器，包含内存访问地址偏移量。

（3）操作伪代码

```
Address = Rn
If  conditionPassed{cond}  then
    If  U = = 1  then
        Rn = Rn + Rm
    Else
```

```
Rn = Rn – Rm
```

（4）说明

① 程序寄存器 r15 被指定为 Rm 或 Rn，指令的执行结果不可预知。

② 如果 Rm 和 Rn 指定为同一寄存器，指令的执行结果不可预知。

4.2.3 批量 Load/Store 指令寻址方式

批量 Load/Store 指令将一片连续内存单元的数据加载到通用寄存器组中或将一组通用寄存器的数据存储到内存单元中。

批量 Load/Store 指令的寻址模式产生一个内存单元的地址范围，指令寄存器和内存单元的对应关系满足这样的规则，即编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中的高地址单元。

指令的语法格式如下。

```
LDM|STM{<cond>}<addressing_mode>  <Rn>{!}，<registers><^>
```

指令的寻址方式如表 4.6 所示。

表 4.6　　批量 Load/Store 指令的寻址方式

	格　　式	模　　式
1	IA（Increment After）	后递增方式
2	IB（Increment Before）	先递增方式
3	DA（Decrement After）	后递减方式
4	DB（Decrement Before）	先递减方式

指令的编码格式如图 4.30 所示。

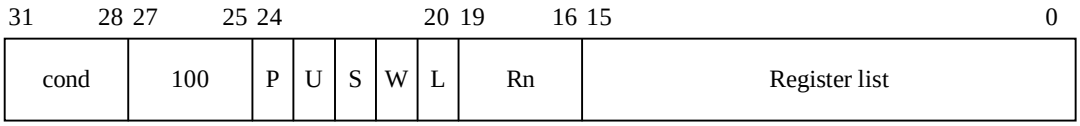


图 4.30　批量 Load/Store 指令编码格式

编码格式中各标志位的含义如表 4.7 所示。

表 4.7　　批量 Load/Store 指令编码格式各标志位含义

位标识	取　　值	含　　义
P	P=0	Rn 包含的地址，是所要访问的内存块的高地址（U=0）还是低地址（U=1）
	P=1	标识 Rn 所指向的内存单元是否被访问
U	U=0	Rn 所指内存单元为所要访问的内存单元块的高地址
	U=1	Rn 所指内存单元为所要访问的内存单元块的低地址

S	S=0	当程序计数器 PC 作为要加载的寄存器之一时，S 标识是否将 spsr 内容拷贝到 cpsr；对于不加载 PC 的 load 指令和所有 store 指令，S 标识特权模式下，使用用户模式寄存器组代替当前模式下寄存器组
	S=1	
W	W=0	数据传送完毕，更新地址寄存器内容
	W=1	
L	L=0	Store 指令
	L=1	Load 指令

1．IA 寻址

（1）编码格式

指令的编码格式如图 4.31 所示。

该寻址方式指定一片连续的内存地址空间，地址空间的大小<address_length>等于寄存器列表中寄存器数目的 4 倍。内存地址范围起始地址<start_address>等于基址寄存器 Rn 的值。结束地址<end_address>等于起始地址<start_address>加上地址空间大小<address_length>。

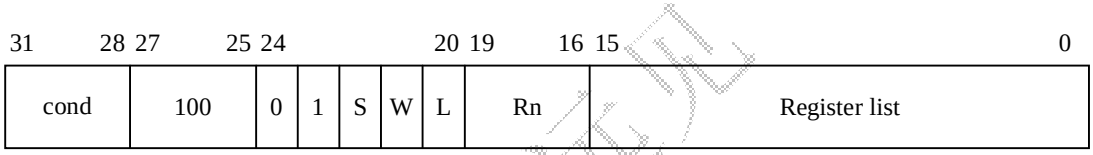


图 4.31　批量 Load/Store 指令——后增加寻址

地址空间中的每个内存单元对应寄存器列表中的一个寄存器。编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中的高地址单元。

当指令执行条件满足并且指令编码格式中 W 位置位，基址寄存器 Rn 的值等于内存地址范围结束地址<end_address>加 4。

（2）语法格式

```
LDM|STM{<cond>}IA  <Rn>{!}，<registers><^>
```

其中：

- IA 标识指令使用“后增加”寻址方式；
- Rn 为基址寄存器，包含内存访问的基地址；
- <registers>为指令操作的寄存器列表；
- <^>表示如果寄存器列表中包含程序计数器 PC，是否将 spsr 拷贝到 cpsr。

（3）操作伪代码

```
Start_address = Rn
End_address = Rn + (Number_of_Set_Bits_In(register_list)*4) – 4
If  conditionPassed(cond) and W = = 1 then
    Rn = Rn + (Number_of_Set_Bits_In(register_list)*4)
```

2．DA 寻址

（1）编码格式

指令的编码格式如图 4.32 所示。

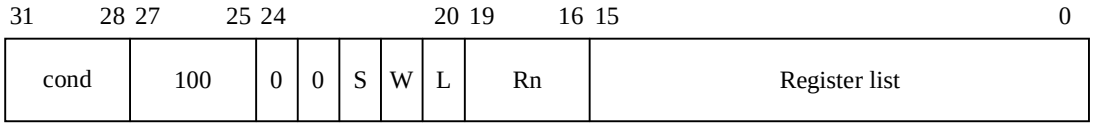


图 4.32　批量 Load/Store 指令——后递减寻址

该寻址方式指定一片连续的内存地址空间，地址空间的大小<address_length>等于寄存器列表中寄存器数目的 4 倍。内存地址范围起始地址<start_address>等于基址寄存器 Rn 的值减去地址空间大小<address_length>并加 4。结束地址<end_address>等于基址寄存器的值。

地址空间中的每个内存单元对应寄存器列表中的一个寄存器。编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中的高地址单元。

当指令执行条件满足并且指令编码格式中 W 位置位时，基址寄存器 Rn 的值等于内存地址范围起始地址<start_address>减 4。

（2）语法格式

```
LDM|STM{<cond>}IA  <Rn>{!}，<registers><^>
```

其中：

- DA 标识指令使用“后递减”寻址方式；
- Rn 为基址寄存器，包含内存访问的基地址；
- <registers>为指令操作的寄存器列表；
- <^>表示如果寄存器列表中包含程序计数器 PC，是否将 spsr 拷贝到 cpsr。

（3）操作伪代码

```
Start_address = Rn – (Number_of_Set_Bits_In(register_list)*4) + 4
End_address = Rn
If  conditionPassed(cond) and W = = 1 then
    Rn = Rn - (Number_of_Set_Bits_In(register_list)*4)
```

3．IB 寻址

（1）编码格式

指令的编码格式如图 4.33 所示。

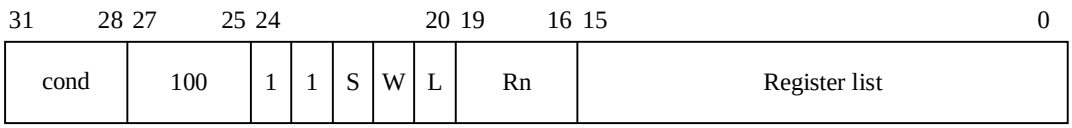


图 4.33　批量 Load/Store 指令——前增加寻址

该寻址方式指定一片连续的内存地址空间，地址空间的大小<address_length>等于寄存器列表中寄存器数目的 4 倍。内存地址范围起始地址<start_address>等于基址寄存器 Rn 的值加 4。结束地址<end_address>等于起始地址<start_address>加上地址空间大小<address_length>。

地址空间中的每个内存单元对应寄存器列表中的一个寄存器。编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中的高地址单元。

当指令执行条件满足并且指令编码格式中 W 位置位，基址寄存器 Rn 的值等于内存地址范围结束地址<end_address>。

（2）语法格式

```
LDM|STM{<cond>}IB  <Rn>{!}，<registers><^>
```

其中：

- IB 标识指令使用“前增加”寻址方式；
- Rn 为基址寄存器，包含内存访问的基地址；
- <registers>为指令操作的寄存器列表；
- <^>表示如果寄存器列表中包含程序计数器 PC，是否将 spsr 拷贝到 cpsr。

（3）操作伪代码

```
Start_address = Rn + 4
End_address = Rn + (Number_of_Set_Bits_In(register_list)*4)
If  ConditionPassed(cond) and W= = 1  then
    Rn = Rn + (Number_Of_Set_Bits_In(register_list)*4)
```

4．DB 寻址

（1）编码格式

指令的编码格式如图 4.34 所示。

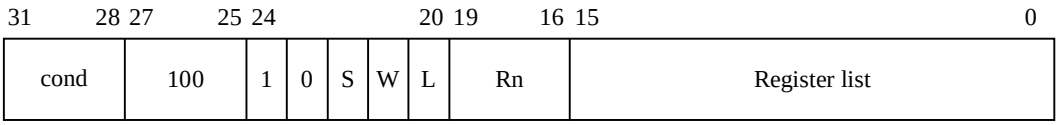


图 4.34　批量 Load/Store 指令——前递减寻址

该寻址方式指定一片连续的内存地址空间，地址空间的大小<address_length>等于寄存器列表中寄存器数目的 4 倍。内存地址范围起始地址<start_address>等于基址寄存器 Rn 的值减地址空间的大小<address_length>。结束地址<end_address>等于基址寄存器的值减 4。

地址空间中的每个内存单元对应寄存器列表中的一个寄存器。编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中的高地址单元。

当指令执行条件满足并且指令编码格式中 W 位置位，基址寄存器 Rn 的值等于内存地址范围起始地址<address_address>。

（2）语法格式

```
LDM|STM{<cond>}DB  <Rn>{!}，<registers><^>
```

其中：

- DB 标识指令使用“前递减”寻址方式；
- Rn 为基址寄存器，包含内存访问的基地址；
- <registers>为指令操作的寄存器列表；
- <^>表示如果寄存器列表中包含程序计数器 PC，是否将 spsr 拷贝到 cpsr。

（3）操作伪代码

```
Start_address = Rn - (Number_Of_Set_Bits_In(register_list)*4)
End_address = Rn - 4
If  ConditionPassed(cond) and W = = 1  then
    Rn = Rn – (Number_Of_Set_Bits_In(register_list)*4)
```

4.2.4 堆栈操作寻址方式

堆栈操作寻址方式和批量 Load/Store 指令寻址方式十分类似。但对于堆栈的操作，数据写入内存和从内存中读出要使用不同的寻址模式，因为进栈操作（pop）和出栈操作（push）要在不同的方向上调整堆栈。

下面详细讨论如何使用合适的寻址方式实现数据的堆栈操作。

根据不同的寻址方式，将堆栈分为以下 4 种。

① Full 栈：堆栈指针指向栈顶元素（last used location）。

② Empty 栈：堆栈指针指向第一个可用元素（the first unused location）。

③ 递减栈：堆栈向内存地址减小的方向生长。

④ 递增栈：堆栈向内存地址增加的方向生长。

根据堆栈的不同种类，将其寻址方式分为以下 4 种。

① 满递减 FD（Full Descending）。

② 空递减 ED（Empty Descending）。

③ 满递增 FA（Full Ascending）。

④ 空递增 EA（Empty Ascending）。

注意 如果程序中有对协处理器数据的进栈/出栈操作，最好使用 FD 或 EA 类型堆栈。这样可以使用一条 STC 或 LDC 指令将数据进栈或出栈。

表 4.8 显示了堆栈的寻址方式和批量 Load/Store 指令寻址方式的对应关系。

表 4.8 堆栈寻址方式和批量 Load/Store 指令寻址方式对应关系

批量数据寻址方式	堆栈寻址方式	L 位	P 位	U 位
LDMDA	LDMFA	1	0	0
LDMIA	LDMFD	1	0	1
LDMDB	LDMEA	1	1	0
LDMIB	LDMED	1	1	1

STMDA	STMED	0	0	0
STMIA	STMEA	0	0	1
STMDB	STMFD	0	1	0
STMIB	STMFA	0	1	1

4.2.5 协处理器 Load/Store 寻址方式

协处理器 Load/Store 指令的语法格式如下。

```
<opcode>{<cond>}{L}  <coproc>，<CRd>，<addressing_mode>
```

表 4.9 显示了该类指令的寻址方式。

表 4.9 协处理器 Load/Store 指令寻址方式

	格 式	说 明
1	[<Rn>,#±<offset_8>*4]	立即数偏移寻址
2	[<Rn>,#±<offset_8>*4]!	前索引立即数偏移寻址
3	[<Rn>],#±<offset_8>*4	后索引立即数偏移寻址
4	[<Rn>], <option>	直接寻址

协处理器 Load/Store 指令的编码方式如图 4.35 所示。
编码格式中各标志位的含义如表 4.10 所示。

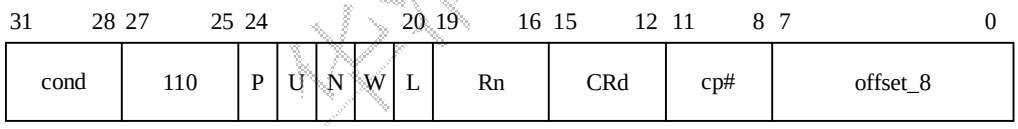


图 4.35 协处理器 Load/Store 指令编码格式

表 4.10 协处理器 Load/Store 指令编码格式各标志位含义

位 标 识	取 值	含 义
P	P=0	标识使用偏移寻址还是前索引寻址（由 W 位决定）
	P=1	标识使用后索引寻址还是直接寻址（由 W 位决定）
U	U=0	从基地址中减去偏移量 offset
	U=1	从基地址中加上偏移量 offset
N	N=0	和具体使用的协处理器相关
	N=1	
W	W=0	指令执行结束，不改变基址寄存器的值
	W=1	访问的内存地址回写到基址寄存器

L	L=0	Store 指令
	L=1	Load 指令

1．[<Rn>,#±<offset_8>*4]

（1）编码格式

指令的编码格式如图 4.36 所示。

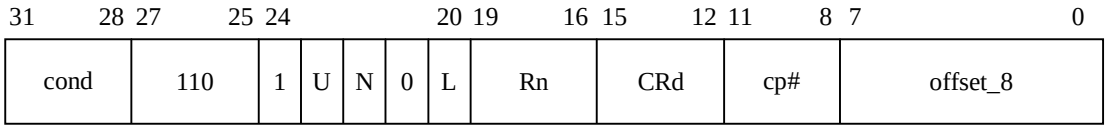


图 4.36　协处理器 Load/Store 指令——立即数寻址

该寻址方式指定一片连续的内存地址空间。访问内存单元的第一个地址<first_addressing>等于基址寄存器<Rn>的值加上/减去指令中寄存器值的 4 倍。接下来的内存访问地址是前一个访问地址加 4。当协处理器发出传输中止信号时，数据传送结束。

这种寻址方式的数据传输数目由协处理器决定。

注意　这种寻址方式最多允许传输 16 的字。

（2）语法格式

```
<opcode>{<cond>}{L}  <coproc>，<CRd>，[<Rn>,#±<offset_8>*4]
```

其中：

- <Rn>为基址寄存器，包含寻址操作的基地址；
- <offset_8>为 8 位立即数，该值的 4 倍为地址偏移量。

（3）操作伪代码

```
If  ConditionPassed(cond)  then
    If  U = = 1 then
        Address = Rn + offset_8 * 4
    Else  /*U = = 0*/
        Address = Rn – offset_8 * 4
    Start_address = address
    While (NotFinished(coprocessor[cp_num]))
        Address = address +4
    End_address = address
```

（4）说明

如果基址寄存器指定为程序计数器 r15，则基地址为当前执行指令地址加 8。

2．[<Rn>,#±<offset_8>*4]!

（1）编码格式

指令的编码格式如图 4.37 所示。

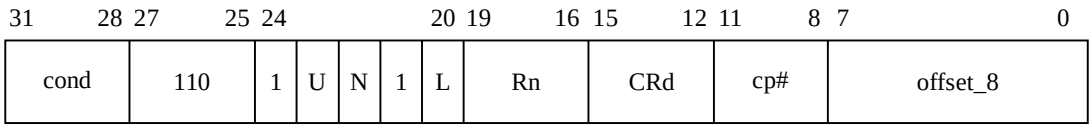


图 4.37　协处理器 Load/Store 指令——前索引立即数寻址

该寻址方式指定一片连续的内存地址空间。访问内存单元的第一个地址<first_addressing>等于基址寄存器<Rn>的值加上/减去指令中寄存器值的 4 倍。如果指令的条件域满足，产生的<first_addressing>回写到基址寄存器 Rn 中。接下来的内存访问地址是前一个访问地址加 4。当协处理器发出传输中止信号时，数据传送结束。

这种寻址方式的数据传输数目由协处理器决定。

注意　这种寻址方式最多允许传输 16 的字。

（2）语法格式

```
<opcode>{<cond>}{L}  <coproc>，<CRd>，[<Rn>,#±<offset_8>*4]!
```

其中：

- <Rn>为基址寄存器，包含寻址操作的基地址；
- <offset_8>为 8 位立即数，该值的 4 倍为地址偏移量；
- ！设置指令编码中的 W 位，更新指令基地址。

（3）操作伪代码

```
If  ConditionPassed(cond)  then
    If  U = = 1 then
        Rn = Rn + offset_8 * 4
    Else  /*U = = 0*/
        Rn = Rn – offset_8 * 4
    Start_address = Rn
    Address = start_address
    While (NotFinished(coprocessor[cp_num]))
        Address = address +4
    End_address = address
```

（4）说明

如果基址寄存器指定为程序计数器 r15，则指令的执行结果不可预知。

3．[<Rn>],#±<Offset_8>*4

（1）编码格式

指令的编码格式如图 4.38 所示。

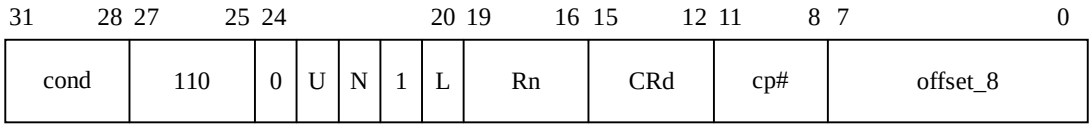


图 4.38　协处理器 Load/Store 指令——后索引立即数寻址

该寻址方式指定一片连续的内存地址空间。访问内存单元的第一个地址<first_addressing>等于基址寄存器<Rn>的值。接下来的内存访问地址是前一个访问地址加 4。当协处理器发出传输中止信号时，数据传送结束。如果指令的条件域满足，Rn 基址寄存器的值更新为 Rn 的值加上/减去 8 位立即数的 4 倍。

这种寻址方式的数据传输数目由协处理器决定。

注意　这种寻址方式最多允许传输 16 的字。

（2）语法格式

```
<opcode>{<cond>}{L}  <coproc>，<CRd>，[<Rn>]，#±<offset_8>*4
```

其中：

- <Rn>为基址寄存器，包含寻址操作的基地址；
- <offset_8>为 8 位立即数，该值的 4 倍为地址偏移量。

（3）操作伪代码

```
If  ConditionPassed(cond)  then
    Start_address = Rn
    If  U = = 1 then
        Rn = Rn + offset_8 * 4
    Else  /*U = = 0*/
        Rn = Rn – offset_8 * 4
    Address = start_address
    While (NotFinished(coprocessor[cp_num]))
        Address = address +4
    End_address = address
```

（4）说明

如果基址寄存器指定为程序计数器 r15，则指令的执行结果不可预知。

4．[<Rn>], <Option>

（1）编码格式

指令的编码格式如图 4.39 所示。

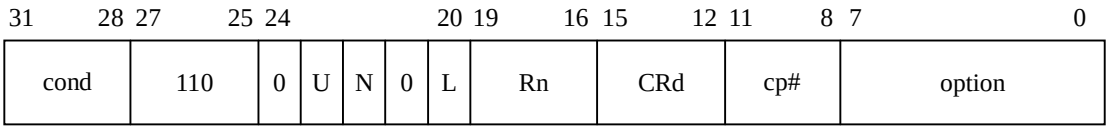


图 4.39　协处理器 Load/Store 指令——直接寻址

该寻址方式指定一片连续的内存地址空间。访问内存单元的第一个地址<first_addressing>等于基址寄存器<Rn>的值。接下来的内存访问地址是前一个访问地址加 4。当协处理器发出传输中止信号时，数据传送结束。

指令不更新基址寄存器的值。指令编码格式中 bits[7：0]保留，所以可以将空闲位用作协处理器指令扩展。

这种寻址方式的数据传输数目由协处理器决定，最多可以传输 16 字。

（2）语法格式

```
<opcode>{<cond>}{L}  <coproc>，<CRd>，[<Rn>]，<Option>
```

其中：

- <Rn>为基址寄存器，包含寻址操作的基地址；
- <option>用作协处理器指令扩展。

（3）操作伪代码

```
If  ConditionPassed(cond)  then
    Start_address = Rn
    Address = start_address
    While (NotFinished(coprocessor[cp_num]))
        Address = address +4
    End_address = address
```

（4）说明

如果基址寄存器指定为程序计数器 r15，则寻址基地址为当前指令地址加 8。

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 5 章 数据传送指令

华清远见

5.1 MOV 指令

1．指令编码格式

MOV 指令是最简单的 ARM 指令，执行的结果就是把一个数 N 送到目标寄存器 Rd，其中 N 可以是寄存器，也可以是立即数。

MOV 指令多用于设置初始值或者在寄存器间传送数据。

指令的编码格式如图 5.1 所示。

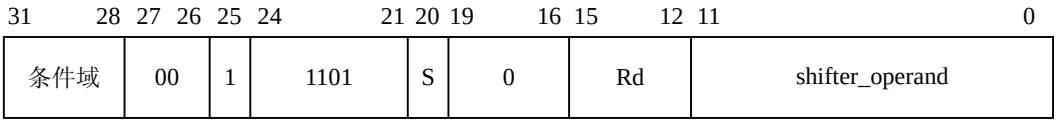


图 5.1 MOV 指令编码格式

MOV 指令将移位码（shifter_operand）表示的数据传送到目的寄存器 Rd，并根据操作的结果更新 CPSR 中相应的条件标志位。

2．指令的语法格式

```
MOV{<cond>}{S}   <Rd>，<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示 MOV 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② S（bit[20]）

如果 S=1，MOV 指令更新 CPSR 中条件标志位的值；如果 S=0，MOV 指令不更新 CPSR 中条件标志位的值。当更新状态寄存器 CPSR 中的条件标志位时，有两种情况。

- 如果指令中的目标寄存器<Rd>不是为 r15，指令根据传送的数值设置 CPSR 中的 N 位和 Z 位（如果数据在传送前需要移位，则根据移位后的数值设置），并根据移位器的进位值设置 CPSR 的 C 位。标志位 V 和其他位不受影响。
- 如果指令中的目标寄存器<Rd>为 r15，则当前处理器模式对应的 SPSR 的值复制到 CPSR 寄存器中，对于用户模式和系统模式，由于没有相应的 SPSR，指令执行的结果不可预知。

③ <Rd>

确定目标寄存器。

④ <shifter_operand>

确定操作数，为目标寄存器传送数据。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   Rd=shifter_operand
```

```
If  S==1 and Rd==r15  then
   CPSR=SPSR
Else  if  S==1  then
    N Flag = Rd[31]
    Z Flag = If  Rd==0  then 1  else 0
    C Flag = shifter_carry_out
    V Flag = unaggected
```

4．指令举例

【例 5.1】MOV 指令

MOV 指令把一个数 N 送到目标寄存器 Rd，其中 N 可以是立即数，也可以是寄存器。

```
MOV     R0, R0            ; R0 = R0... NOP 指令
MOV     R0, R0, LSL#3     ; R0 = R0 * 8
```

如果 r15 是目的寄存器，将修改程序计数器或标志。这用于返回到调用代码，方法是把连接寄存器的内容传送到 r15。

```
MOV     PC, R14           ; 退出到调用者
MOVS    PC, R14           ; 退出到调用者并恢复标志位
```

5．指令的使用

MOV 指令主要完成以下功能。

- 将数据从一个寄存器传送到另一个寄存器。
- 将一个常数值传送到寄存器中。
- 实现无算术和逻辑运算的单纯移位操作，操作数乘 2^n 可以用左移 n 位来实现。
- 当 PC 寄存器（r15）用作目的寄存器时，可以实现程序跳转。如“MOV　PC，LR”，所以这种跳转可以实现子程序调用以及从子程序返回，代替指令“B，BL”。

注意 在体系结构 v4 和 v5 以上的版本，必须使用 BX 指令代替 MOV　PC，LR 指令，因为 BX 指令可以自动实现 ARM 和 Thumb 的切换。

- 当 PC 寄存器作为目标寄存器且指令中 S 位被设置时，指令在执行跳转操作的同时，将当前处理器模式的 SPSR 寄存器内容复制到 CPSR 中。这种指令“MOVS PC LR”可以实现从某些异常中断中返回。

5.2　MVN 指令

1．指令编码格式

MVN 是反相传送（Move Negative）指令。它将操作数的反码传送到目的寄存器。

MVN 指令多用于向寄存器传送一个负数或生成位掩码。

指令的编码格式如图 5.2 所示。

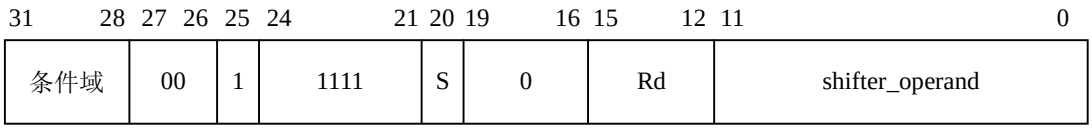


图 5.2 MVN 指令编码格式

MVN 指令将<shifter_operand>表示的数据的反码传送到目的寄存器 Rd。并根据操作的结果更新 CPSR 中相应的条件标志位。

2．指令的语法格式

```
MNV{<cond>}{S}   <Rd>，<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示 MVN 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② S（bit[20]）

如果 S=1，MVN 指令更新 CPSR 中条件标志位的值；如果 S=0，MVN 指令不更新 CPSR 中条件标志位的值。当更新状态寄存器 CPSR 中的条件标志位时，有两种情况。

- 如果指令中的目标寄存器<Rd>不是为 r15，指令根据传送的数值设置 CPSR 中的 N 位和 Z 位（如果数据在传送前需要移位，则根据移位后的数值设置），并根据移位器的进位值设置 CPSR 的 C 位。标志位 V 和其他位不受影响。
- 如果指令中的目标寄存器<Rd>为 r15，则当前处理器模式对应的 SPSR 的值复制到 CPSR 寄存器中，对于用户模式和系统模式，由于没有相应的 SPSR，指令执行的结果不可预知。

③ <Rd>

确定目标寄存器。

④ <shifter_operand>

确定操作数，为目标寄存器传送数据。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   Rd = NOT(shifter_operand)
   If  S==1 and Rd==r15  then
     CPSR = SPSR
   Else  if  S==1  then
      N Flag = Rd[31]
      Z Flag = If  Rd==0  then 1  else 0
      C Flag = shifter_carry_out
      V Flag = unaggected
```

4．指令举例

【例 5.2】MVN 指令

MVN 指令和 MOV 指令相同也可以把一个数 N 送到目标寄存器 Rd，其中 N 可以是立即数，也可以是寄存器。

注意 这是逻辑非操作而不是算术操作，这个取反的值加 1 才是它的取负的值。

```
MVN     R0, #4              ; R0 = -5
MVN     R0, #0              ; R0 = -1
```

5．指令的使用

MVN 指令主要完成以下功能。

- 向寄存器中传送一个负数。
- 生成位掩码（bit mask）。
- 求一个数的反码。

5.3 单寄存器的 Load/Store 指令

Load/Store 内存访问指令在 ARM 寄存器和存储器之间传送数据。ARM 指令中有 3 种基本的数据传送指令。

① 单寄存器 Load/Store 指令（Single Register）

这些指令在 ARM 寄存器和存储器之间提供更灵活的单数据项传送方式。数据项可以是字节、16 位半字或 32 位字。

② 多寄存器 Load/Store 内存访问指令

这些指令的灵活性比单寄存器传送指令差，但可以使大量的数据更有效地传送。它们用于进程的进入和退出、保存和恢复工作寄存器以及拷贝存储器中的一块数据。

③ 单寄存器交换指令（Single Register Swap）

这些指令允许寄存器和存储器中的数值进行交换，在一条指令中有效地完成 Load/Store 操作。它们在用户级编程中很少用到。它的主要用途是在多处理器系统中实现信号量（Semaphores）的操作，以保证不会同时访问公用的数据结构。

5.3.1 字数据传送指令

这种指令用于把单一的数据传入或者传出一个寄存器。支持的数据类型有字节（8 位）、半字（16 位）和字（32 位）。

表 5.1 总结了所有单寄存器的 Load/Store 指令。

表 5.1 单寄存器 Load/Store 指令

指　令	作　用	操　作
LDR	把一个字装入一个寄存器	Rd←mem32[address]
STR	将存储器中的字保存到寄存器	Rd→mem32[address]
LDRB	把一个字节装入一个寄存器	Rd←mem8[address]
STRB	将寄存器中的低 8 位字节保存到存储器	Rd→mem8[address]
LDRH	把一个半字装入一个寄存器	Rd←mem16[address]
STRH	将寄存器中的低 16 位半字保存到存储器	Rd→mem16[address]
LDRBT	用户模式下将一个字节装入寄存器	Rd←mem8[address] under user mode
STRBT	用户模式下将寄存器中的低 8 位字节保存到存储器	Rd→mem8[address] under user mode
LDRT	用户模式下把一个字装入一个寄存器	Rd←mem32[address]under user mode
STRT	用户模式下将存储器中的字保存到寄存器	Rd→mem32[address]]under user mode
LDRSB	把一个有符号字节装入一个寄存器	Rd←sign{mem8[address]}
LDRSH	把一个有符号半字装入一个寄存器	Rd←sign{mem16[address]}

1．LDR 指令

（1）指令编码格式

LDR 指令用于从内存中将一个 32 位的字读取到目标寄存器。

指令的编码格式如图 5.3 所示。

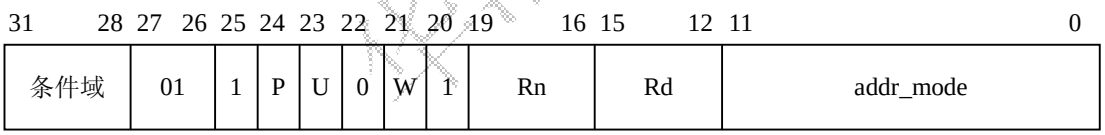


图 5.3　LDR 指令编码格式

LDR 指令根据<addr_mode>所确定的地址模式将一个 32 位字读取到指令中的目标寄存器<Rd>。如果指令中的寻址方式确定的地址不是字对齐的，则读出的数值要进行循环右移。所移位数为寻址方式确定的地址 bits[1：0]8 的倍，也就是说处理器将取到的数值作为字的最低位处理。

如果设置了 L 位，则进行装载，否则进行存储。

如果设置了 P 位，则使用预先变址寻址，否则使用过后变址寻址。

如果设置了 U 位，则给出的偏移量被加到基址寄存器上，否则从中减去偏移量。

如果设置了 B 位，传送内存的一个字节，否则传送一个字。这在助记符末尾添加后缀“B”，如 MOV r7，r5 变为 MOVB r7，r5。

W 位的解释依赖于使用的地址模式。

- 对于预先变址寻址，设置 W 位强制把它用做地址转换的最终地址写回基址寄存器中（例如，传送的副作用是 Rn:= Rn +/-offset。这在汇编器中表示为给指令加上后缀 “!”。)。

- 对于过后变址寻址，地址总是写回，设置 W 位指示在进行传送之前强制地址转换。这在汇编器中表示为给指令加上后缀“T”。

当 PC 作为 LDR 的目的寄存器<Rd>时，从存储器取得的数据将被当作目标地址值，程序将跳转到目标地址开始执行。

（2）指令的语法格式

```
LDR{<cond>}  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDR 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。



```
If  ConditionPassed{cond}  then
   If  address[1:0] == 0b00  then
       Value = Memory[address,4]
Else  if  address[1:0] == 0b01  then
    Value = Memory[address,4]  Rotate_Right 8
Else  if  address[1:0] == 0b10  then
    Value = Memory[address,4]  Rotate_Right 16
Eles  /* address[1:0] == 0b11*/
    Value = Memory[address,4]  Rotate_Right 24
If  (Rd is R15)  then
    If (architecture version 5 or above)  then
           PC = value AND 0xfffffffe
           T Bit = value[0]
    Else
           PC = value AND 0xfffffffc
Else
    Rd = value
```

（4）指令举例

```
LDR  r1，[r0，#0x12]      ;将 r0+12 地址处的数据读出，保存到 r1 中（r0 的值不变）
LDR  r1，[r0]             ;将 r0 地址处的数据读出，保存到 r1 中（零偏移）
LDR  r1，[r0，r2]         ;将 r0+r2 地址的数据读出，保存到 r1 中（r0 的值不变）
LDR  r1，[r0，r2，LSL #2] ;将 r0+r2×4 地址处的数据读出，保存到 r1 中（r0，r2 的值不变）
LDR  Rd，label            ;label 为程序标号，label 必须是当前指令的±4KB 范围内
```

```
    LDR  Rd，[Rn]，＃0x04      ;Rn 的值用作传输数据的存储地址。在数据传送后，将偏移量 0x04 与
                                Rn 相加，结果写回到 Rn 中。Rn 不允许是 r15
```

注意 地址对齐问题：大多数情况下，必须保证用于 32 位传送的地址是 32 位对齐的。

2．STR 指令

（1）指令编码格式

STR 指令用于将一个 32 位的字数据写入到指令中指定的内存单元。

指令的编码格式如图 5.4 所示。

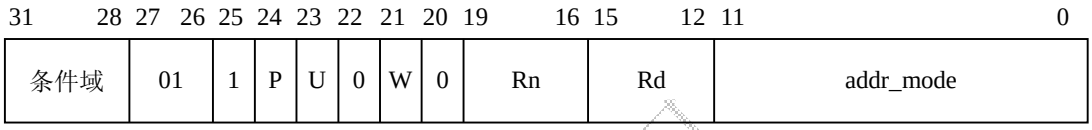


图 5.4　STR 指令编码格式

（2）指令的语法格式

```
    STR{<cond>}  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 STR 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
    If  ConditionPassed{cond}  then
        Memory[address，4]=Rd
```

（4）指令举例

LDR/STR 指令用于对内存变量的访问、内存缓冲区数据的访问、查表、外围部件的控制操作等等，若使用 LDR 指令加载数据到 PC 寄存器，则实现程序跳转功能，这样也就实现了程序散转。

① 变量访问

```
NumCount  EQU  0x40003000      ;定义变量 NumCount
LDR  R0，=NumCount            ;使用 LDR 伪指令装载 NumCount 的地址到 R0
LDR  R1，[R0]                 ;取出变量值
ADD  R1，R1，＃1              ;NumCount=NumCount+1
STR   R1，[R0]                ;保存变量
```

② GPIO 设置

```
GPIO—BASE  EQU   0xe0028000   ;定义 GPIO 寄存器的基地址
……
LDR    R0,=GPIO—BASE
LDR    R1,=0x00ffff00         ;将设置值放入寄存器
STR    R1,[R0,#0x0C]          ;IODIR=0x00ffff00，IOSET 的地址为 0xE0028004
```

③ 程序散转

```
…
MOV  r2,r2,LSL #2             ;功能号乘以 4，以便查表
LDR  PC,[PC,r2]               ;查表取得对应功能子程序地址，并跳转
NOP
FUN—TAB  DCD   FUN—SUB0
         DCD   FUN—SUB1
         DCD   FUN—SUB2
           …
```

5.3.2 字节数据传送指令（LDRB/STRB）

1．LDRB 指令

（1）指令编码格式

LDRB 指令根据<addr_mode>所确定的地址模式将一个 8 位字节读取到指令中的目标寄存器<Rd>。

指令的编码格式如图 5.5 所示。

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
条件域	01	I	P	U	1	W	1	Rn	Rd	addr_mode

图 5.5　LDRB 指令编码格式

注意　LDRB 指令加载一个内存地址的 8 位字节到一个通用寄存器中。寄存器的高位数据补 0。

（2）指令的语法格式

```
LDR{<cond>}B  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDRB 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
if  ConditionPassed{cond}  then
   Rd = Memory[address,1]
```

注意 当 PC 作为位基地址出现在指令中时，指令中将会使用 PC 相关地址，使用这种方法可以编写自己的位置无关（position-independ）指令。

2．STRB 指令

（1）指令编码格式

STRB 指令从寄存器中取出指定的 8 位字节放入寄存器的低 8 位，并将寄存器的高位补 0。

指令的编码格式如图 5.6 所示。

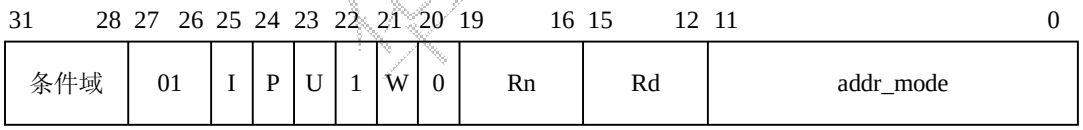


图 5.6　STRB 指令编码格式

（2）指令的语法格式

```
STR{<cond>}B  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 STRB 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

注意 当 PC 作为目标寄存器<Rd>出现在指令中时，指令的执行结果不可预知。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
if  ConditionPassed{cond}  then
   Memory[address,1] = Rd[7:0]
```

5.3.3 半字数据传送指令（LDRH/STRH）

1. LDRH 指令

（1）指令编码格式

LDRH 指令用于从内存中将一个 16 位的半字读取到目标寄存器。

如果指令的内存地址不是半字节对齐的，指令的执行结果不可预知。

指令的编码格式如图 5.7 所示。

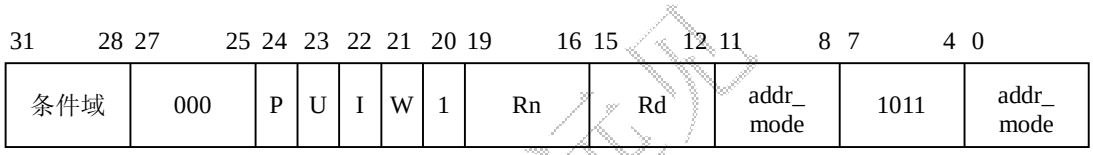


图 5.7 LDRH 指令的编码格式

（2）指令的语法格式

```
LDR{<cond>}H  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDRH 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

注意 如果 PC 作为目标寄存器，指令的执行结果不可预知。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
if  ConditionPassed{cond}  then
   if address[0]==0
       data=Memory[address,2]
   else  /*address[0]==1*/
```

```
        data=unpredictable
    Rd=data
```

注意 在包含系统控制协处理器的芯片应用中，如果定义了地址对齐检测，当 bit[0]!=0 时，将发生地址对齐异常。

2．STRH 指令

（1）指令编码格式

STRH 指令从寄存器中取出指定的 16 位半字放入寄存器的低 16 位，并将寄存器的高位补 0。

指令的编码格式如图 5.8 所示。

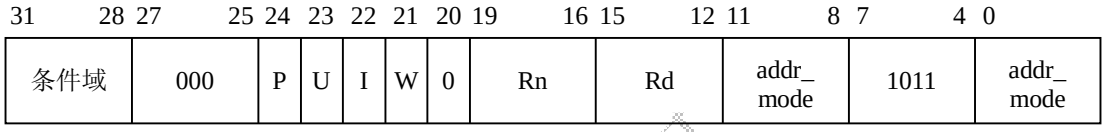


图 5.8　STRH 指令的编码格式

（2）指令的语法格式

```
STR{<cond>}H  <Rd>，<addr_mode>
```

① <cond>

指令编码中的条件域。它指示 STRH 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

注意 如果 PC 作为目标寄存器，指令的执行结果不可预知。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
if  ConditionPassed{cond}  then
    if address[0]==0
        data=Rd[15:0]
    else  /*address[0]==1*/
        data=unpredictable
    Memory[address,2]=data
```

5.3.4 用户模式字数据传送指令（LDRT/STRT）

1．LDRT 指令

（1）指令编码格式

LDRT 指令用于从内存中将一个 32 位的字读取到目标寄存器。

指令的编码格式如图 5.9 所示。

LDRT 指令根据<addr_mode>所确定的地址模式将一个 32 位字读取到指令中的目标寄存器<Rd>。如果指令中的寻址方式确定的地址不是字对齐的，则读出的数值要进行循环右移。所移位数为寻址方式确定的地址 bits[1：0]的 8 倍。也就是说处理器将取到的数值作为字的最低位处理。

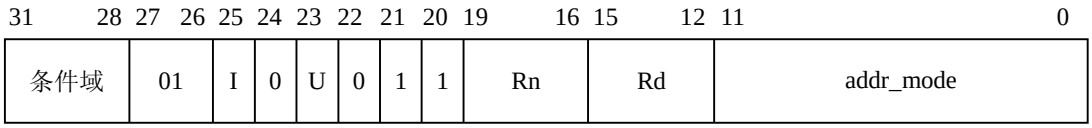


图 5.9 LDRT 指令编码格式

当处理器在特权模式下使用此指令时，内存系统将该操作当作一般用户模式下得内存访问指令。

注意 指令的编码格式中，P 位指定位 “0”，也就是说 LDRT 指令的寻址方式为固定寻址方式，即后索引编码寻址（post_indexed_addressing_mode）。

（2）指令的语法格式

```
LDR{<cond>}T  <Rd>，<post_indexed_addressing_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDRT 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <post_indexex_address_mode>

使用后索引地址模式寻址。

注意 后索引地址模式中 P=0 并且 W=0（即 bit[21]=0、bit[24]=0）。但此指令 P=0 并且 W=1（即 bit[21]=1、bit[24]=0）。但实际的寻址操作是一样的。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   If  address[1:0]==0b00
        Rd=Memory[address,4]
```

```
    Else  if  address[1:0]==0b01
          Rd=Memory[address,4]  Rotate_Right  8
    Else  if  address[1:0]==0b10
          Rd=Memory[address,4]  Rotate_Right  16
    Else  address[1:0]==0b11
          Rd=Memory[address,4]  Rotate_Right  24
```

2．STRT 指令

（1）指令编码格式

STRT 指令用于将一个 32 位的字数据写入到指令中指定的内存单元。

当处理器在特权模式下执行此指令时，内存系统将该操作当作一般用户模式下的内存访问操作。

指令的编码格式如图 5.10 所示。

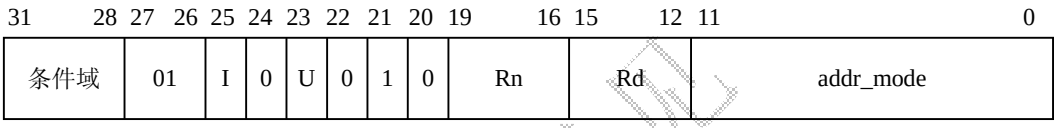


图 5.10　STR 指令编码格式

（2）指令的语法格式

```
STR{<cond>}T  <Rd>，<post_indexed_addressing_mode>
```

① <cond>

为指令编码中的条件域。它指示 STRT 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <post_indexed_address_mode>

使用后索引地址模式寻址，参见 LDRT 指令。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Memory[address,4]=Rd
```

5.3.5　用户模式字节数据传送指令（LDRBT/STRBT）

1．LDRBT 指令

（1）指令编码格式

LDRBT 指令根据<post_indexed_addressing_mode>地址模式将一个 8 位字节读取到指令

中的目标寄存器<Rd>。

当处理器在特权模式下执行此指令时，内存系统将该操作当作一般用户模式下的内存访问操作。

指令的编码格式如图 5.11 所示。

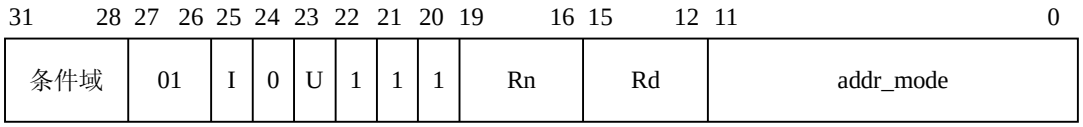


图 5.11　LDRBT 指令编码格式

注意　LDRBT 指令加载一个内存地址的 8 位字节到一个通用寄存器中。寄存器的高位数据补 0。

（2）指令的语法格式

```
LDR{<cond>}BT  <Rd>，<post_indexed_addressing_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDRBT 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <post_indexed_addressing_mode>

使用后索引地址模式寻址，参见 LDRT 指令。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   Rd=Memory[address,1]
```

2．STRBT 指令

（1）指令编码格式

STRBT 指令用于将一个 8 位的字节数据写入到指令中指定的内存单元。

当处理器在特权模式下执行此指令时，内存系统将该操作当作一般用户模式下的内存访问操作。

指令的编码格式如图 5.12 所示。

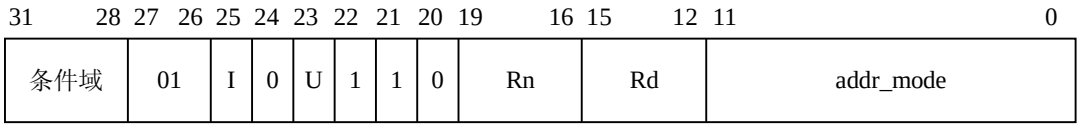


图 5.12　STRBT 指令编码格式

（2）指令的语法格式

```
STR{<cond>}BT  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDRBT 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <post_indexed_addressing_mode>

使用后索引地址模式寻址，参见 LDRT 指令。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Memory[address，1]=Rd[7:0]
```

5.3.6 有符号的字节/半字数据传送指令（LDRBT/STRBT）

1．LDRSB 指令

（1）指令编码格式

LDRSB 指令根据<addr_mode>所确定的地址模式将一个 8 位字节读取到指令中的目标寄存器<Rd>。

注意 LDRSB 与 LDRB 指令的不同之处在于它将寄存器的高 24 位设置成该字节数据的符号位的值（即将该 8 位字节数据进行符号位扩展，生成 32 位字数据）。

指令的编码格式如图 5.13 所示。

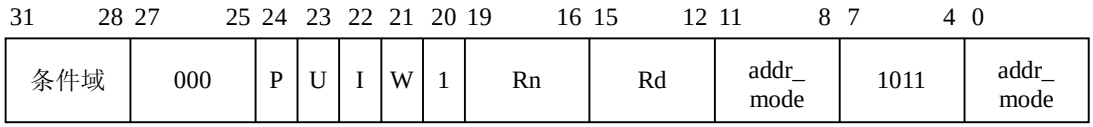


图 5.13 LDRSB 指令编码格式

（2）指令的语法格式

```
LDR{<cond>}SB  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDRSB 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
If  ConditionPassed{cond}  then
   data=Memory[address,1]
Rd=SignExtend{data}
```

2．LDRSH 指令

（1）指令编码格式

LDRSH 指令根据<addr_mode>所确定的地址模式将一个 16 位半字读取到指令中的目标寄存器<Rd>。

注意 LDRSH 与 LDRH 指令的不同之处在于它将寄存器的高 16 位设置成该字节数据的符号位的值（即将该 16 位字节数据进行符号位扩展，生成 32 位字数据）。

指令的编码格式如图 5.14 所示。

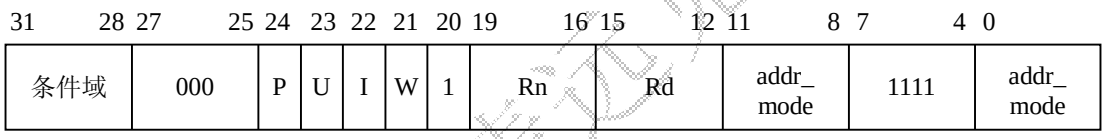


图 5.14　LDRSH 指令编码格式

（2）指令的语法格式

```
LDR{<cond>}SH  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDRSH 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
If  ConditionPassed{cond}  then
   if  address[0]==0
      data=Memory[address,2]
  else  /*address[0] ==1*/
      data=UNPREDICTABLE
```

```
Rd=SignExtend{data}
```

5.4 多寄存器 Load/Store 内存访问指令

多寄存器 Load/Store 内存访问指令也叫批量加载/存储指令，它可以实现在一组寄存器和一块连续的内存单元之间传送数据。LDM 用于加载多个寄存器，STM 用于存储多个寄存器。多寄存器 Load/Store 内存访问指令允许一条指令传送 16 个寄存器的任何子集或所有寄存器。

多寄存器 Load/Store 内存访问指令主要用于现场保护、数据复制和参数传递等。

注意 多寄存器 Load/Store 内存访问指令会增加中断延时，因为 ARM 通常不会打断正在执行的指令去响应中断，而必须等到指令执行完。也就是说，如果一个中断在多寄存器 Load/Store 内存访问指令执行期间产生，那么处理器在多寄存器 Load/Store 内存访问指令执行完后才对中断响应。

表 5.2 总结了多寄存器 Load/Store 内存访问指令

表 5.2 多寄存器 Load/Store 内存访问指令

指　令	作　用	操　作
LDM	装载多个寄存器	{Rd}*^{N}←mem32[start address+4*N]
STM	保存多个寄存器	{Rd}*^{N}→mem32[start address+4*N]

5.4.1 多寄存器内存字数据传送指令

1．LDM（1）指令

（1）指令编码格式

LDM（1）指令将数据从连续的内存单元中读取到指令中指定的寄存器列表中的各寄存器中。

当 PC 包含在 LDM 指令的寄存器列表中时，指令从内存中读取的字数据将被作为目标地址值，指令执行后程序将从目标地址处开始执行，从而实现了指令的跳转。

指令的编码格式如图 5.15 所示。

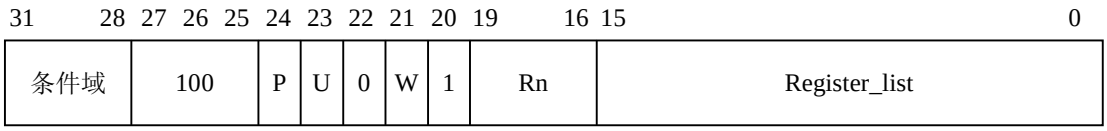


图 5.15　LDM（1）指令编码格式

（2）指令的语法格式

```
LDM{<cond>}<addressing_mode>  <Rn>{!},  <registers>
```

① <cond>

为指令编码中的条件域。它指示 LDM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P、U 和 W 位。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ ！

设置指令编码格式中的W位。它使指令执行后将操作数的内存地址写入基址寄存器<Rn>中；如果！被忽略，W 位为 0，指令执行完后，不修改基址寄存器的值。

注意 如果基址寄存器包含在指令列表中，当指令执行完后，基址寄存器的值是新加载进的特定内存地址的值。也就是说，即使指令没有出现在指令列表中，基址寄存器的值也可能被修改。

⑤ <registers>

被加载的寄存器列表。不同的寄存器之间用“，”隔开。完整的寄存器列表包含在“{}”中。编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中高地址单元。

注意 无论寄存器在寄存器列表“{}”中如何排列，都将遵循上述规则。

寄存器 r0～r15 分别对应于指令编码中 bit[0]～bit[15]位。如果 Ri 存在于寄存器列表中，则相应的位等于 1，否则为 0。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPass{cond}  then
    Address=start_address

    For  i=0  to 14
       If  register_list[i]==1  then
          Ri=Memory[address,4]
          Address=address+4

    If  register_list[15]==1  then
           Value = Memory[address,4]
           If(architecture version 5 or above)  then
                Pc= value  AND  0xfffffffe
```

```
                T bit=value[0]
            Else
                Pc= value  AND  0xfffffffc
            Address=address+4
    Assert end_address=address+4
```

2．STM（1）指令

（1）指令编码格式

STM（1）指令将指令中寄存器列表中的各寄存器数值写入到连续的内存单元中。主要用于块数据的写入、数据栈操作以及进入子程序时保存相关寄存器的操作。

指令编码格式如图 5.16 所示。

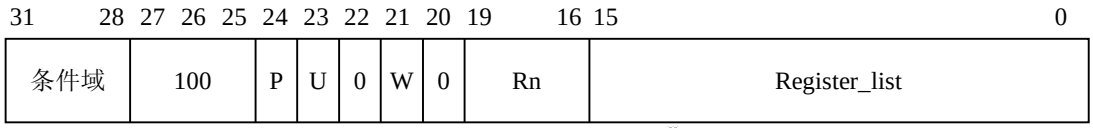


图 5.16　STM（1）指令编码格式

（2）指令的语法格式

```
STM{<cond>}<addressing_mode>  <Rn>{!},  <registers>
```

① <cond>

为指令编码中的条件域。它指示 STM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P、U 和 W 位。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ ！

设置指令编码格式中的W位。它使指令执行后将操作数的内存地址写入基址寄存器<Rn>中；如果！被忽略，W 位为 0，指令执行完后，不修改基址寄存器的值。

⑤ <registers>

被加载的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中高地址单元。

寄存器 r0～r15 分别对应于指令编码中 bit[0]～bit[15]位。如果 Ri 存在于寄存器列表中，则相应的位等于 1，否则为 0。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
```

```
Address=Start_address
For  i=0  to  15
     If  register_list[i]==1
     Memory[address,4]=Ri
     Address=address+4
Assert  end_address==address-4
```

5.4.2 用户模式多寄存器内存字数据传送指令

1．LDM（2）指令

（1）指令编码格式

LDM（2）指令将数据从连续的内存单元中读取到指令中指定的寄存器列表中的各寄存器中。

注意 与 LDM（1）指令不同，PC 不能包含在寄存器列表中。

指令的编码格式如图 5.17 所示。

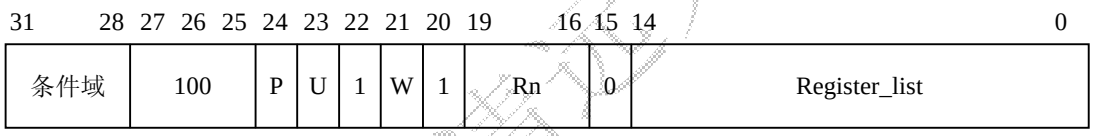


图 5.17 LDM（2）指令编码格式

（2）指令的语法格式

```
LDM{<cond>}<addressing_mode>  <Rn>,  <registers_without_pc>^^
```

① <cond>

为指令编码中的条件域。它指示 LDM（2）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P 位和 U 位。此指令中 W 位指定为 0。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ <registers_without_pc>^

被加载的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。此寄存器列表中不能包含 PC 寄存器。

如果 PC 包含在寄存器列表中，指令的执行结果不可预知。

其他细节可参考 LDM（1）指令。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Address=start_address
    For  i=0  to  14
        If  register_list[i]==1
             Ri_usr=Memory[address,4]
             Address=address+4
    Assert  end_address == address-4
```

2．STM（2）指令

（1）指令编码格式

STM（2）指令将指令中寄存器列表中的各寄存器数值写入到连续的内存单元中。主要用于块数据的写入、数据栈操作以及进入子程序时保存相关寄存器等操作。

指令编码格式如图 5.18 所示。

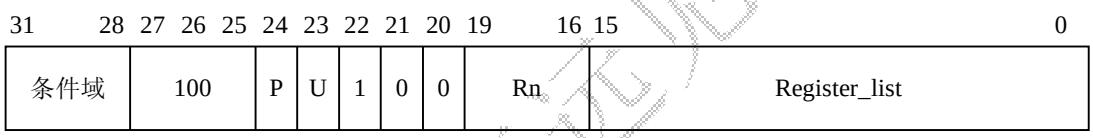


图 5.18　STM（2）指令编码格式

（2）指令的语法格式

```
STM{<cond>}<addressing_mode>  <Rn>,  <registers >^
```

① <cond>

为指令编码中的条件域。它指示 LDM（2）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P 位和 U 位。此指令中 W 位指定为 0。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ <registers >^

寄存器列表。只能使用用户模式下的寄存器。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Address=start_address
    For  i=0  to  15
```

```
            If  register_list[i] == 1
                 Memory[address,4]=Ri_usr
                 Address = address +4
        Assert  end_address == address-4
```

5.4.3 带状态寄存器的多寄存器内存字数据装载指令（LDM（3））

（1）指令编码格式

LDM（3）指令将数据从连续的内存单元中读取数据到寄存器列表中的各寄存器中。它同时将当前处理器模式对应的 SPSR 寄存器的内容复制到 CPSR 寄存器中。

当 PC 包含在 LDM 指令的寄存器列表中时，指令从内存中读取的数据将被作为目标地址值，指令执行后程序将从目标地址处开始执行，从而实现了指令的跳转。

在 ARM v5 及以上的版本和 T 系列的 ARM v4 版本中，SPSR 寄存器的 T 位将复制到 CPSR 寄存器的 T 位，该位决定目标地址处的程序状态。在以前的版本中程序继续执行在 ARM 状态。

指令的编码格式如图 5.19 所示。

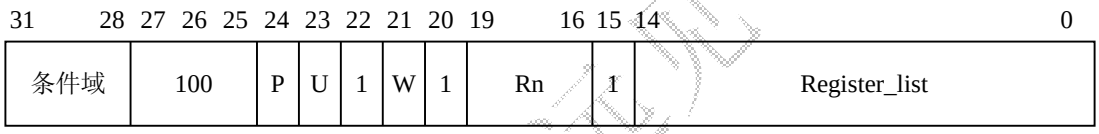


图 5.19 LDM（3）指令编码格式

（2）指令的语法格式

```
LDM{<cond>}<addressing_mode>  <Rn>{!},  <registers_and_pc>^
```

① <cond>

为指令编码中的条件域。它指示 LDM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P、U 和 W 位。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ ！

设置指令编码格式中的W位。它使指令执行后将操作数的内存地址写入基址寄存器<Rn>中；如果！被忽略，W 位为 0，指令执行完后，不修改基址寄存器的值。

注意 如果基址寄存器包含在指令列表中，当指令执行完后，基址寄存器的值是新加载进的特定内存地址的值。也就是说，即使指令没有出现在指令列表中，基址寄存器的值也可能被修改。

⑤ <registers_and_pc>^

寄存器列表。

注意 在本格式的指令中寄存器列表中必须包含 PC 寄存器。

被加载的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。编号低的寄存器对应于内存中的低地址单元，编号高的寄存器对应于内存中的高地址单元。

寄存器 r0～r15 分别对应于指令编码中 bit[0]～bit[15]位。如果 Ri 存在于寄存器列表中，则相应的位等于 1，否则为 0。

该指令执行时将当前处理器模式下的 SPSR 值复制到 CPSR 中。指令的其他参数可参见 LDM（1）指令格式。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPass{<cond>}  then
    Address=start_address
    For i=0  to  14
        If  register_list[i]==1 then
            Ri=Memory[address,4]
            Address=address+4
    CPSR=SPSR
    Value=memory[address,4]
    If {architecture version 4T, 5 or above} and {T bit ==1} then
    Else
        Pc=value AND oxfffffffc
    Address=address + 4
    Assert end_address=address-4
```

5.4.4 数据传送指令应用

LDM/STM 批量加载/存储指令可以实现在一组寄存器和一块连续的内存单元之间传输数据。LDM 为加载多个寄存器，STM 为存储多个寄存器。允许一条指令传送 16 个寄存器的任何子集或所有寄存器。指令格式如下：

```
LDM{cond}<模式>  Rn{!}，regist{^}
STM{cond}<模式>  Rn{!}，regist{^}
```

LDM/STM 的主要用途有现场保护、数据复制和参数传递等。其模式有 8 种，如下所示。前面 4 种用于数据块的传输，后面 4 种是堆栈操作。

（1）IA：每次传送后地址加 4。

（2）IB：每次传送前地址加 4。

（3）DA：每次传送后地址减 4。

（4）DB：每次传送前地址减 4。

（5）FD：满递减堆栈。

（6）ED：空递增堆栈。

（7）FA：满递增堆栈。

（8）EA：空递增堆栈。

其中，寄存器 Rn 为基址寄存器，装有传送数据的初始地址，Rn 不允许为 R15；后缀“!”表示最后的地址写回到 Rn 中；寄存器列表 reglist 可包含多于一个寄存器或寄存器范围，使用“,”分开，如{R1，R2，R6～R9}，寄存器排列由小到大排列；“^”后缀不允许在用户模式下，只能在系统模式下使用。若在 LDM 指令用寄存器列表中包含有 PC 时使用，那么除了正常的多寄存器传送外，将 SPSR 拷贝到 CPSR 中，这可用于异常处理返回；使用“^”后缀进行数据传送且寄存器列表不包含 PC 时，加载/存储的是用户模式寄存器，而不是当前模式寄存器。

注意 地址对齐问题，在这些指令中，忽略地址位[1:0]。

批量加载/存储指令举例如下。

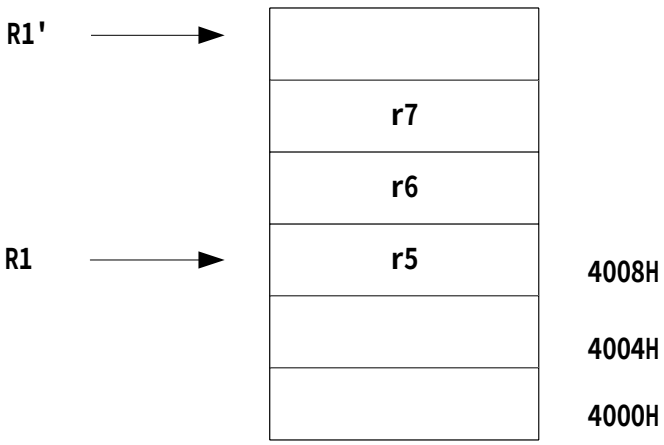
```
LDMIA  r0!，{r3~r9}       ;加载 r0 指向的地址上的多字数据，保存到 r3～r9 中，r0 值更新
STMIA  r1!，{r3～r9}      ;将 r3～r9 的数据存储到 r1 指向的地址上，r1 值更新
STMFD  SP!，{r0～r7，LR}  ;现场保存，将 r0～r7、LR 入栈
LDMFD  SP!，{r0～r7，PC}^ ;恢复现场，异常处理返回
```

在进行数据复制时，先设置好源数据指针，然后使用块拷贝寻址指令 LDMIA/STMIA、LDMIB/STMIB、LDMDA/STMDA、LDMDB/STMDB 进行读取和存储。而进行堆栈操作时，则要先设置堆栈指针，一般使用 SP 然后使用堆栈寻址指令 STMFD/LDMFD、STMED/LDMED、STMEA/LDMEA 实现堆栈操作。

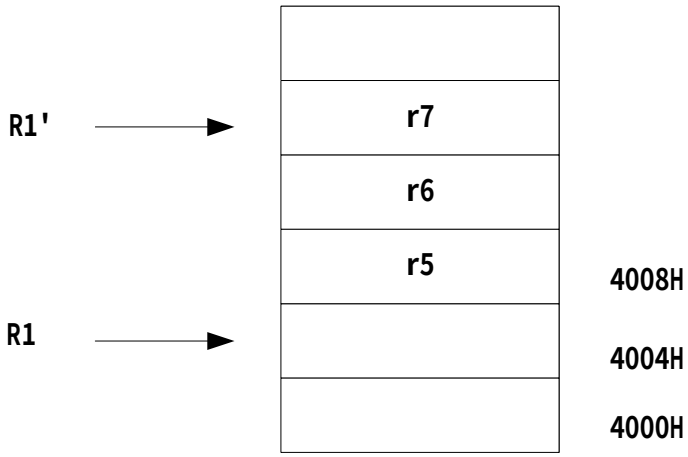
多寄存器传送指令如例 5.3 所示。其中 r1 为指令执行前的基址寄存器，r1’则为指令执行后的基址寄存器。

【例 5.3】多寄存器传送指令示意。

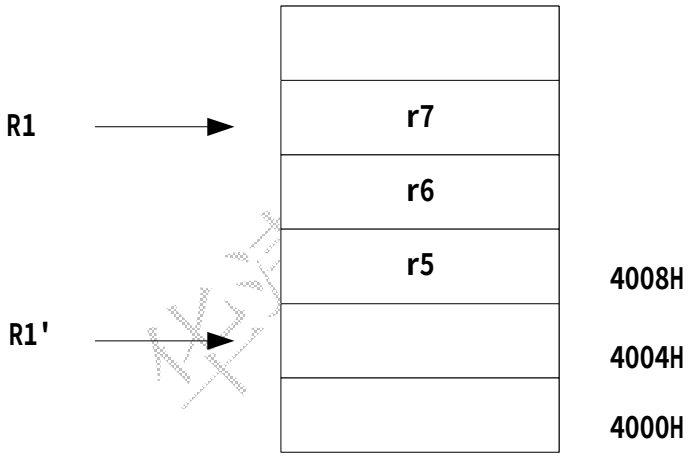
（1）STMIA　r1，{r5～r7}



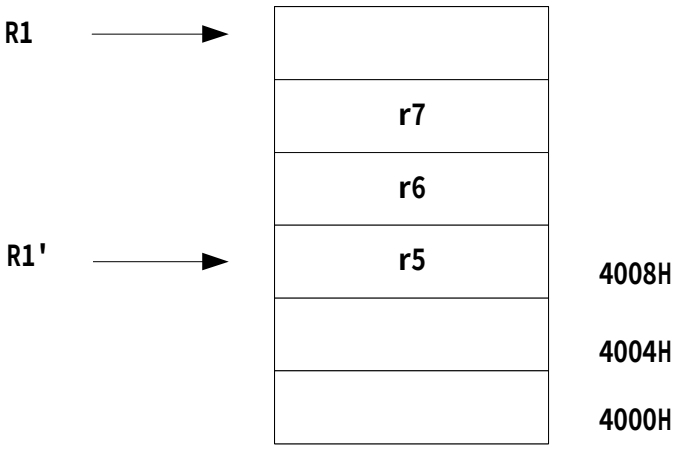
（2）STMIB r1!，{r5～r7}



（3）STMDA r1!，{r5～r7}



（4）STMDB r1!，{r5～r7}



数据是存储在基址寄存器的地址之上还是之下，地址是存储第一个值之前还是之后、增加还是减少，如表 5.3 所示。

表 5.3　　多寄存器 Load/Store 内存访问指令映射

		向 上 生 长		向 下 生 长	
		满	空	满	空
增加	之前	STMIB			LDMIB
		STMFA			LDMED
	之后		STMIA	LDMIA	
			STMEA	LDMFD	
增加	之前		LDMDB	STMDB	
			LDMEA	STMFD	
	之后	LDMDA			STMDA
		LDMFA			STMED

【例 5.4】使用 LDM/STM 进行数据复制。

```
LDR  r0，=SrcData        ;设置源数据地址
LDR  r1，=DstData        ;设置目标地址
LDMIA   r0，{r2～r9}     ;加载 8 字数据到寄存器 r2～r9
STMIA   r1，{r2～r9}     ;存储寄存器 r2～r9 到目标地址
```

【例 5.5】使用 LDM/STM 进行现场寄存器保护，常在子程序或异常处理使用。

```
SENDBYTE
     STMFD    SP!，{r0～r7，LR}   ;寄存器压栈保护
   ……
     BL       DELAY               ;调用 DELAY 子程序
   ……
     LDMFD    SP!，{r0～r7，PC}   ;恢复寄存器，并返回
```

5.5　单数据交换指令

交换指令是 load/store 指令的一种特例，它把一个寄存器单元的内容与寄存器内容交换。交换指令是一个原子操作（atomic operation），也就是说，在连续的总线操作中读/写一个存储单元，在操作期间阻止其他任何指令对该存储单元的读/写。

交换指令如表 5.4 所示。

表 5.4　　交换指令 SWP

指　令	作　用	操　作

SWP	字交换	Tmp=men32[Rn] Mem32[Rn]=Rm Rd=tmp

续表

指　令	作　用	操　作
SWPB	字节交换	Tmp=men8[Rn] Mem8[Rn]=Rm Rd=tmp

注意 交换指令在执行期间不能被其他任何指令或其他任何总线访问打断，在此期间系统占用总线（holds the bus），直至交换完成。

5.5.1 字交换指令 SWP

（1）指令编码格式

SWP 指令用于将内存中的一个字单元和一个指定寄存器的值相交换。操作过程如下，假设内存单元地址存放在寄存器<Rn>中，指令将<Rn>中的数据读取到目的寄存器 Rd 中，同时将另一个寄存器<Rm>的内容写入到该内存单元中。当<Rd>和<Rm>为同一个寄存器时，指令交换该寄存器和内存单元的内容。

指令的编码格式如图 5.20 所示。

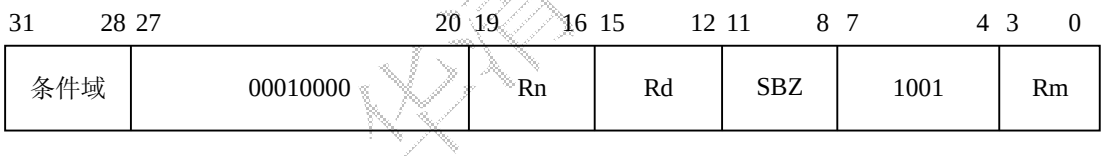


图 5.20　SWP 指令编码格式

（2）指令的语法格式

```
SWP{<cond>}  <Rd>，<Rm>，[<Rn>]
```

① <cond>

为指令编码中的条件域。它指示 SWP 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

目标寄存器。

③ <Rm>

寄存器包含将要存储到内存中的数据。

④ <Rn>

寄存器中包含将要访问的内存地址。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    If  Rn[1:0]==0b00  then
          Temp=memory[Rn,4]
    Else  if  Rn[1:0]==0b01  then
          Temp=memory[Rn,4]  Rotate_right 8
    Else  if  Rn[1:0]==0b10  then
          Temp=memory[Rn,4]  Rotate_right 16
    Else  /*  Rn[1:0]==0b11  */
          Temp=memory[Rn,4]  Rotate_right 24
    Memory[Rn,4]=Rm
    Rd=temp
```

5.5.2　字节交换指令 SWPB

（1）指令编码格式

SWPB 指令用于将内存中的一个字节单元和一个指定寄存器的低 8 位值相交换，操作过程如下。假设内存单元地址存放在寄存器<Rn>中，指令将<Rn>中的数据读取到目的寄存器 Rd 中，寄存器 Rd 的高 24 位设为 0，同时将另一个寄存器<Rm>的低 8 位内容写入到该内存字节单元中。当<Rd>和<Rm>为同一个寄存器时，指令交换该寄存器低 8 位内容和内存字节单元的内容。

指令的编码格式如图 5.21 所示。

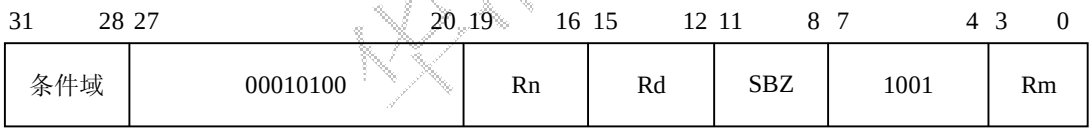


图 5.21　SWPB 指令编码格式

（2）指令的语法格式

```
SWP{<cond>}B  <Rd>，<Rm>，[<Rn>]
```

① <cond>

为指令编码中的条件域。它指示 SWPB 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

目标寄存器。

③ <Rm>

寄存器包含将要存储到内存中的数据。

④ <Rn>

寄存器中包含将要访问的内存地址。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Temp=Memory[Rn,1]
    Memory[Rn,1]=Rm[7:0]
    Rd=temp
```

5.5.3 交换指令 SWP 应用

寄存器和存储器交换指令 SWP 指令用于将一个内存单元（该单元地址放在寄存器 Rn 中）的内容读取到一个寄存器 Rd 中，同时将另一个寄存器 Rm 的内容写到该内存单元中，使用 SWP 可实现信号量操作。

格式如下。

```
SWP{cond}B  Rd，Rm，[Rn]
```

其中，B 为可选后缀，若有 B，则交换字节，否则交换 32 位字。Rd 为目的寄存器，存储从存储器中加载的数据，同时，Rm 中的数据将会被存储到存储器中。若 Rm 与 Rn 相同，则为寄存器与存储器内容进行交换。Rn 为要进行数据交换的存储器地址，Rn 不能与 Rd 和 Rm 相同。

【例 5.6】SWP 指令举例。

```
SWP  r1，r1，[r0]       ;将 r1 的内容与 r0 指向的存储单元内容进行交换
SWPB  r1，r2，[r0]      ;将 r0 指向的存储单元内容读取一字节数据到 r1 中（高 24 位清零），
                         并将 r2 的内容写入到该内存单元中（最低字节有效）
```

使用 SWP 指令可以方便地进行信号量操作。

```
12C_SEM     EQU     0x40003000
            ……
12C_SEM_WAIT
        MOV     r0,#0
        LDR     r0,=12C_SEM
        SWP     r1,r1,[r0]    ;取出信号量，并将其设为 0
        CMP     r1,#0         ;判断是否有信号
        BEQ     12C_SEM_WAIT  ;若没有信号则等待
```

5.6 程序状态寄存器指令

ARM 指令集提供了两条指令，可直接控制程序状态寄存器（PSR，Program State Register）。

MRS 指令用于把 CPSR 或 SPSR 的值传送到一个寄存器；MSR 与之相反，把一个寄存器的内容传送到 CPSR 或 SPSR。这两条指令结合，可用于对 CPSR 和 SPSR 进行读/写操作。

交换指令如表 5.5 所示。

表 5.5　　程序状态寄存器指令

指　令	作　用	操　作
MRS	把程序状态寄存器的值送到一个通用寄存器	Rd=SPR
MSR	把通用寄存器的值送到程序状态寄存器或把一个立即数送到程序状态字	PSR[field]=Rm 或 PSR[field]=immediate

在指令语法中可看到一个称为 fields 的项，它可以是控制（C）、扩展（X）、状态（S）及标志（F）的组合。

注意　程序不能通过直接修改 CPSR 中的 T 位控制直接将程序状态切换到 Thumb 状态，必须通过 BX 等指令完成程序状态的切换。

5.6.1　MRS

（1）指令编码格式

MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。

指令的编码格式如图 5.22 所示。

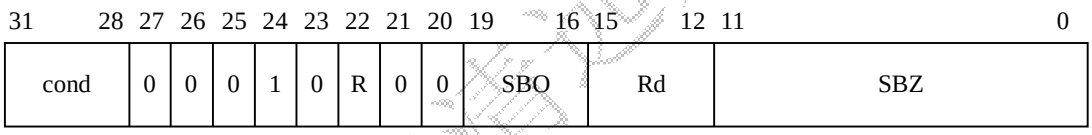


图 5.22　MRS 指令编码格式

当数据被移到通用寄存器以后，就可以对数据进行处理。

（2）指令的语法格式

```
MRS{<cond>}  <Rd>，CPSR
MRS{<cond>}  <Rd>，SPSR
```

① <cond>

为指令编码中的条件域。它指示 LDM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

目标寄存器。当 r15 被用作目标寄存器时，指令执行结果不可预知。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    If  R==1  then
        Rd=SPSR
    ELSE
```

```
Rd=CPSR
```

注意
① 当操作码 opcode[11：0]≠0x000 时，指令的执行结果不可预知。
② 当操作码 opcode[19：16]≠0b1111 时，指令的执行结果不可预知。
③ 当在用户模式下对 SPSR 进行操作时，指令的执行结果不可预知。

5.6.2 MSR

（1）指令编码格式

MSR 指令用于将通用寄存器中的内容或立即数传送到程序状态寄存器中。因此指令的编码格式也有两种格式。

指令的源操作数为通用寄存器时编码格式如图 5.23 所示。

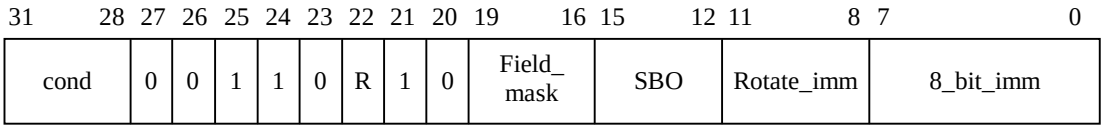


图 5.23 源操作数为通用寄存器的 MSR 指令编码

指令的源操作数为立即数时编码格式如图 5.24 所示。

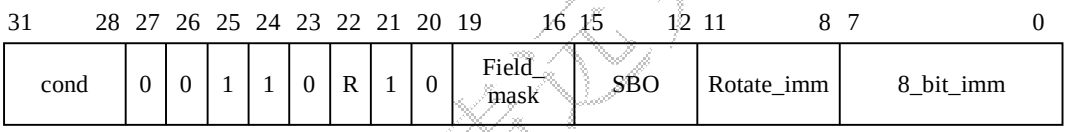


图 5.24 源操作数为立即数的 MSR 指令编码

（2）指令的语法格式

```
MSR{<cond>}  CPSR_<fields>，#<immediate>
MSR{<cond>}  CPSR_<fields>，#<Rm>
MSR{<cond>}  SPSR_<fields>，#<immediate>
MSR{<cond>}  SPSR_<fields>，#<Rm>
```

① <cond>

为指令编码中的条件域。它指示 LDM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <fields>

设置状态寄存器中需要操作的位。状态寄存器的 32 位可以分为 4 个 8 位的域（field）。

- bits[31：24]为条件标志位域，用 f 表示。
- bits[23：16]状态位域，用 s 表示。
- bits[15：8]扩展位域，用 x 表示。
- bits[7：0]控制位域，用 c 表示。

③ < immediate >

表示将要传送到状态寄存器中的立即数。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   If  opcode[25]==1
        Operand=8_bit_immediate  Rotate_Right {rotate_imm*2}
   Else  /*opcode[25]==0*/
        Operand=Rm
   If  R==0  then
        If  field_mask[0]==1  and  InAPrivilegedMode()  then
             CPSR[7:0]=operand[7:0]
        If  field_mask[1]==1  and  InAPrivilegedMode()  then
             CPSR[15:8]=operand[15:8]
        If  field_mask[2]==1  and  InAPrivilegedMode()  then
             CPSR[23:16]=operand[23:16]
        If  field_mask[2]==1  then
             CPSR[31:24]=operand[31:24]
   Else  /*R==1*/
        If  field_mask[0]==1  and  InAPrivilegedMode()  then
             SPSR[7:0]=operand[7:0]
        If  field_mask[1]==1  and  InAPrivilegedMode()  then
             SPSR[15:8]=operand[15:8]
        If  field_mask[2]==1  and  InAPrivilegedMode()  then
             SPSR[23:16]=operand[23:16]
        If  field_mask[2]==1  then
             SPSR[31:24]=operand[31:24]
```

5.6.3　程序状态寄存器指令应用

在 ARM 处理器中，只有 MRS 指令可以将状态寄存器 CPSR 或 SPSR 读出到通用寄存器中。指令格式如下：

```
MRS{cond}  Rd，PSR
```

其中，Rd 为目标寄存器，Rd 不允许为程序计数器 PC。PSR 为 CPSR 或 SPSR。

MRS 指令举例如下。

```
MRS  r1，CPSR    ;将 CPSR 状态寄存器读取，保存到 r1 中
MRS  r2，SPSR    ;将 SPSR 状态寄存器读取，保存到 r1 中
```

MRS 指令读取 CPSR，可用来判断 ALU 的状态标志以及 IRQ/FIQ 中断是否允许等；在异常处理程序中，读 SPSR 可指定进入异常前的处理器状态等。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读-修改-写操作，可用来进行处理器模式切换，允许/禁止 IRQ/FIQ

中断等设置。另外，进程切换或允许异常中断嵌套时，也需要使用 MRS 指令读取 SPSR 状态值并保存起来。

【例 5.7】使能 IRQ 中断

```
ENABLE_IRQ
        MRS    r0，CPSR
        BIC     r0，r0，＃0x80
        MSR    CPSR_c，r0
        MOV    PC，LR
```

【例 5.8】禁止 IRQ 中断

```
DISABLE_IRQ
       MRS   r0,CPSR
       ORR   r0,r0,#0x80
       MSR   CPSR_c,r0
       MOV   PC,LR
```

在 ARM 处理器中，只有 MSR 指令可以直接设置状态寄存器 CPSR 或 SPSR。指令格式如下。

```
MSR{cond}  PSR_field,#immed_8r
MSR{cond}  PSR_field,Rm
```

其中，PSR 是指 CPSR 或 SPSR。<fields>设置状态寄存器中需要操作的位。状态寄存器的 32 位可以分为 4 个 8 位的域（field）。bits[31：24]为条件标志位域，用 f 表示；bits[23：16]为状态位域，用 s 表示；bits[15：8]为扩展位域，用 x 表示；bits[7：0]为控制位域，用 c 表示；immed_8r 为要传送到状态寄存器指定域的立即数，8 位；Rm 为要传送到状态寄存器指定域的数据源寄存器。

MSR 指令举例如下。

```
MSR  CPSR_c，＃0xD3          ;CPSR[7:0]=0xD3，切换到管理模式
MSR  CPSR_cxsf，r3           ;CPSR=R3
```

只有在特权模式下才能修改状态寄存器。

程序中不能通过 MSR 指令直接修改 CPSR 中 T 位控制位来实现 ARM 状态/Thumb 状态的切换，必须使用 BX 指令来完成处理器状态的切换（因为 BX 指令属转移指令，它会打断流水线状态，实现处理器状态的切换）。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读-修改-写操作，可用来进行处理器模式切换以及允许/禁止 IRQ/FIQ 中断等设置。

【例 5.9】堆栈指令初始化

```
INITSTACK
         MOV   r0,LR              ;保存返回地址
;设置管理模式堆栈
```

```
        MSR   CPSR_c,#0xD3
        LDR   SP,StackSvc
;设置中断模式堆栈
        MSR   CPSR_c,#0xD2
        LDR   SP,StackSvc
```

华清远见

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 6 章 数据处理指令

华清远见

数据处理指令是指对存放在寄存器中的数据进行处理的指令。主要包括算术指令、逻辑指令、比较与测试指令以及乘法指令。

如果在数据处理指令前使用 S 前缀，指令的执行结果将会影响 CPSR 中的标志位。

数据处理指令如表 6.1 所示。

表 6.1 数据处理指令列表

操 作 码	助 记 符	操 作	行 为
0000	AND	逻辑加	Rd：=Rn AND op2[1]
0001	EOR	逻辑异或	Rd：=Rn EOR op2
0010	SUB	减	Rd：=Rn − op2
0011	RSB	翻转减	Rd：=op2 − Rn
0100	ADD	加	Rd：=Rn + op2
0101	ADC	带进位的加	Rd：=Rn + op2 + C
0110	SBC	带进位的减	Rd：=Rn− op2 + C − 1
0111	RSC	带进位的翻转减	Rd：=op2 − Rn + C − 1
1000	TST	测试	Rn AND op2 并更新标志位
1001	TEQ	测试相等	Rn EOR op2 并更新标志位
1010	CMP	比较	Rn−op2 并更新标志位
1011	CMN	负数比较	Rn+op2 并更新标志位
1100	ORR	逻辑或	Rd：=Rn OR op2
1110	BIC	位清 0	Rd：=Rn AND NOT（op2）

指令操作的伪代码如下面程序段所示。

```
<opcode2>{<cond>}  <Rn>,<shifter_operand>
<opcode2>:=CMP|CMN|TST|TEQ
<opcode3>{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
<opcode3>:=ADD|SUB|RSB|ADC|SBC|RSC|AND|BIC|EOR|ORR
```

指令的编码格式如图 6.1。

cond	00	I	opcode	S	Rn	Rd	shifter_operand

[1] op2 即为指令中的第二个操作数。ARM 数据处理指令使用 3 地址格式，这就意味着分别指定两个源操作数和一个目的寄存器。第一个源操作数总是目的寄存器，第二个源操作数又叫移位操作数（a shifter operand），它可能是寄存器、移位后的寄存器或立即数。第二个操作数如果是寄存器，则应用于它的移位可能是逻辑或算术移位，或是循环移位。移位的位数可以由立即数指定，也可以由第 4 个寄存器指定。

图 6.1　数据处理指令的编码格式

I：区分第二操作数是立即数或寄存器。

S：标志指令的条件域是否更新 CPSR。

Rn：指示第一源操作数寄存器。

Rd：指示目的寄存器。

shifter_operand：指示第二源操作数。

6.1　AND 逻辑与指令

1．指令编码格式

AND 指令将<shifter_operand>表示的数值与寄存器<Rn>的值按位（bitwise）做逻辑与操作，并将结果保存到目标寄存器<Rd>中，同时根据操作的结果更新 CPSR 寄存器。

指令的编码格式如图 6.2 所示。

cond	00	I	0000	S	Rn	Rd	shifter_operand

图 6.2　ADD 指令的编码格式

2．指令的语法格式

```
AND{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

S 位（bit[20]）决定指令的执行是否影响 CPSR 中的条件域。当 S 位清 0 时，指令执行不影响 CPSR。当 S 位置位时（并且不是 r15），则有以下规则。

- 如果结果为负，则标志位 N 置位；否则清 0（也就是说 N 等于结果的第 31 位）。
- 如果结果为 0，则标志位 Z 置位；否则清 0。
- 当操作定义为算术操作（ADD、ADC、SUB、SBC、RSB 或 RSC）时，标志位 C 设置为 ALU 的进位输出；否则设置为移位器的进位输出。如果不需要移位，则保持 C。
- 在非算术操作中，标志位 V 保持原值。在算术操作中，如果有从第 30 位到第 31 位的溢出，则置位；如果不发生溢出，则清 0。仅当算术操作中操作数被认为是 2 的补码的有符号数时，这个标志位才有意义，而且指示结果超出范围。

若指令中的目标寄存器<Rd>为 r15，则当前处理器模式对应的 SPSR 的值被复制到 CPSR 寄存器中，对应用户模式和系统模式，由于没有相应的 SPSR，指令的执行结果不可预知。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd=Rn AND shifter_operand
    If  S==1  and  Rd==r15  then
    CPSR=SPSR
    Else  if  S==1  then
         N flag=Rd[31]
         Z flag=if  Rd==0  then  1  else  0
         C flag=shifter_carry_out
         V flag=unaggected
```

4．指令举例

【例 6.1】AND 指令举例。

（1）AND 的真值表（二者都是 1 则结果为 1）如表 6.2 所示。

表 6.2　　AND 指令真值表

Op_1	Op_2	结　果
0	0	0
0	1	0
1	0	0
1	1	1

（2）保留 R0 中的 0 位和 1 位，丢弃其余的位。

```
AND    R0, R0, #3          ;
```

（3）R2=R1&R3

```
AND   R2,R1,R3           ;
```

（4）R0=R0&0x01,取出最低位数据

```
ANDS   R0,R0,#0x01          ;
```

6.2 EOR 逻辑异或指令

1．指令的编码格式

逻辑异或 EOR（Exclusive OR）指令将寄存器<Rn>中的值和<shifter_operand>的值执行按位“异或”操作，并将执行结果存储到目的寄存器<Rd>中，同时根据指令的执行结果更新 CPSR 中相应的条件标志位。

指令的编码格式如图 6.3 所示。

cond	00	I	0001	S	Rn	Rd	shifter_operand

图 6.3　EOR 指令的编码格式

2．指令的语法格式

```
EOR{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

S 位（bit[20]）决定指令的执行是否影响 CPSR 中的条件域。当 S 位清 0 时，指令执行不影响 CPSR。当 S 位置位时（并且不是 r15），则有以下规则。

- 如果结果为负，则标志位 N 置位；否则清 0（也就是说 N 等于结果的第 31 位）。
- 如果结果为 0，则标志位 Z 置位；否则清 0。
- 当操作定义为算术操作（ADD、ADC、SUB、SBC、RSB 或 RSC）时，标志位 C 设置为 ALU 的进位输出；否则设置为移位器的进位输出。如果不需要移位，则保持 C。
- 在非算术操作中，标志位 V 保持原值。在算术操作中，如果有从第 30 位到第 31 位的溢出，则置位；如果不发生溢出，则清 0。仅当算术操作中操作数被认为是 2 的补码的有符号数时，这个标志位才有意义，而且指示结果超出范围。

若指令中的目标寄存器<Rd>为 r15，则当前处理器模式对应的 SPSR 的值被复制到 CPSR 寄存器中。用户模式和系统模式下，由于没有相应的 SPSR，指令的执行结果不可预知。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

注意 当指令的编码格式中 I 位等于 0，并且移位操作数 shifter_operand 中 bit[7]和 bit[4]也都等于 1，则指令并非 EOR 指令。详情请参阅 ARM 系统结构参考手册。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd=Rn EOR shifter_operand
    If  S==1  and  Rd==r15  then
    CPSR=SPSR
    Else  if  S==1  then
         N flag=Rd[31]
         Z flag=if  Rd==0  then  1  else  0
         C flag=shifter_carry_out
         V flag=unaggected
```

4．指令举例

【例 6.2】EOR 指令举例。

（1）EOR 的真值表（二者不同则结果为 1）如表 6.3 所示。

表 6.3 EOR 指令真值表

Op_1	Op_2	结　果
0	0	0
0	1	1
1	0	1
1	1	0

（2）反转 R0 中的位 0 和 1

```
EOR     R0, R0, #3            ;
```

（3）将 R1 的低 4 位取反

```
EOR     R1,R1,#0x0F;
```

（4）R2=R1∧R0

```
EOR     R2,R1,R0;
```

（5）将 R5 和 0x01 进行逻辑异或，结果保存到 R0，并根据执行结果设置标志位。

```
EORS     R0，R5，＃0x01;
```

6.3 SUB 减操作指令

1．指令的编码格式

SUB（Subtract）减操作指令，从寄存器<Rn>中减去<shifter_operand>表示的数值，并将结果保存到目标寄存器<Rd>中，并根据指令的执行结果设置 CPSR 中相应的标志位。

指令的编码格式如图 6.4 所示。

cond	00	I	0010	S	Rn	Rd	shifter_operand

图 6.4　EOR 指令的编码格式

2．指令的语法格式

```
SUB{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

S 位（bit[20]）决定指令的执行是否影响 CPSR 中的条件域。当 S 位清 0 时，指令执行不影响 CPSR。当 S 位置位时（并且不是 r15），则有以下规则。

- 如果结果为负，则标志位 N 置位；否则清 0（也就是说 N 等于结果的第 31 位）。
- 如果结果为 0，则标志位 Z 置位；否则清 0。
- 当操作定义为算术操作（ADD、ADC、SUB、SBC、RSB 或 RSC）时，标志位 C 设置为 ALU 的进位输出；否则设置为移位器的进位输出。如果不需要移位，则保持 C。
- 在非算术操作中，标志位 V 保持原值。在算术操作中，如果有从第 30 位到第 31 位的溢出，则置位；如果不发生溢出，则清 0。仅当算术操作中操作数被认为是 2 的补码的有符号数时，这个标志位才有意义，而且指示结果超出范围。

若指令中的目标寄存器<Rd>为 r15，则当前处理器模式对应的 SPSR 的值被复制到 CPSR 寄存器中。用户模式和系统模式下，由于没有相应的 SPSR，指令的执行结果不可预知。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

注意 在操作时要特别小心 C 标志位的设置。当 C=1 时，表示减操作没有借位发生；当 C=0 时，说明减操作发生了借位。也就是说 C 标志位在这里被用作了非借位标志。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd=Rn – shifter_operand
    If  S==1  and  Rd==r15  then
    CPSR=SPSR
    Else  if  S==1  then
        N flag=Rd[31]
        Z flag=if  Rd==0  then  1  else  0
        C flag=NOT BorrowForm{Rn – shifter_operand}
        V flag=OverflowFrom{Rn – shifter_operand}
```

4．指令举例

【例 6.3】SUB 指令举例。

（1）R0=R1-R2

```
SUB    R0, R1, R2
```

（2）R0=R1-256

```
SUB    R0, R1, #256
```

（3）R0=R2-(R3<<1)

```
SUB    R0, R2, R3,LSL#1
```

6.4 RSB 减翻转指令

1．指令的编码格式

RSB（Reverse Subtract）减操作指令，从寄存器<shifter_operand>中减去<Rn>表示的数值，并将结果保存到目标寄存器<Rd>中，并根据指令的执行结果设置 CPSR 中相应的标志位。

指令的编码格式如图 6.5 所示。

cond	00	I	0011	S	Rn	Rd	shifter_operand

图 6.5　RSB 指令的编码格式

2．指令的语法格式

```
RSB{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

详见 SUB 指令。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

注意　当指令的编码格式中 I 位等于 0，并且移位操作数 shifter_operand 中 bit[7]和 bit[4]也都等于 1，则指令并非 RSB 指令。详情请参阅 ARM 系统结构参考手册。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd= shifter_operand－Rn
    If  S==1  and  Rd==r15  then
    CPSR=SPSR
    Else  if  S==1  then
        N flag=Rd[31]
        Z flag=if  Rd==0  then  1  else  0
        C flag=NOT BorrowFrom{shifter_operand - Rn}
        V flag=OverflowFrom{shifter_operand - Rn}
```

4．指令举例

【例 6.4】RSB 指令举例。

下面指令序列可以求一个 64 位数值的负数。64 位数放在寄存器 R0 与 R1 中，其负数放在 R2 和 R3 中。其中 R0 与 R2 中放低 32 位值。

```
RSBS    R2，R0，＃0;
RSC     R3，R1，＃0;
```

6.5 ADD 加操作指令

1．指令的编码格式

ADD 加操作指令，将寄存器<shifter_operand>的值加上<Rn>表示的数值，并将结果保存到目标寄存器<Rd>中，并根据指令的执行结果设置 CPSR 中相应的标志位。

指令的编码格式如图 6.6 所示。

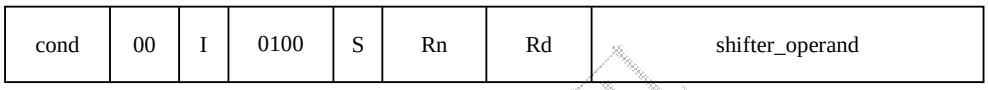


图 6.6 ADD 指令的编码格式

2．指令的语法格式

```
ADD{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

决定指令的操作是否影响 CPSR 中条件标志位的值。当有 S 位时指令更新 CPSR 中条件标志位的值；当没有 S 位时指令不更新 CPSR 中条件标志位的值。当有 S 位时，有下面两种情况。

- 如果<Rd>不是 R15，则 CPSR 中的 N 和 Z 位根据指令的执行结果设置。C 根据指令操作是否产生一个进位（即一个无符号溢出）来设置；V 位根据是否有带符号的溢出来设置。CPSR 中的其他位不受影响。
- 如果<Rd>是程序计数器 R15，则当前程序状态的 SPSR 拷贝到 CPSR。如果处理器处于用户模式或系统模式，则指令的执行结果不可预知。因为这两种模式没有自己的私有 SPSR 寄存器。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）

位和 shifter_operand（bits[11：0]）位。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd=Rn + shifter_operand
    If  S==1  and  Rd==r15  then
        CPSR=SPSR
    Else  if  S==1  then
        N flag=Rd[31]
        Z flag=if  Rd==0  then  1  else  0
        C flag=CarryFrom{ Rn + shifter_operand }
        V flag=OverflowFrom{ Rn + shifter_operand }
```

4．指令举例

【例 6.5】ADD 指令举例。

```
ADD     R0, R1, R2          ; R0 = R1 + R2
ADD     R0, R1, #256        ; R0 = R1 + 256
ADD     R0, R2, R3,LSL#1    ; R0 = R2 + (R3 << 1)
```

6.6　ADC 带进位的加法指令

1．指令的编码格式

ADC 加操作指令，将寄存器<shifter_operand>的值加上<Rn>表示的数值，再加上 CPSR 中的 C 条件标志位的值，将结果保存到目标寄存器<Rd>中，并根据指令的执行结果设置 CPSR 中相应的标志位。

指令的编码格式如图 6.7 所示。

cond	00	I	0101	S	Rn	Rd	shifter_operand

图 6.7　ADC 指令的编码格式

2．指令的语法格式

```
ADC{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

S 位（bit[20]）决定指令的操作是否影响 CPSR 中条件标志位的值。当 S=1 时指令更新 CPSR 中条件标志位的值；当 S=0 时指令不更新 CPSR 中条件标志位的值。当 S=1 时，有下面两种情况。

- 如果<Rd>不是 R15，CPSR 中的 N 和 Z 位根据指令的执行结果设置。C 位根据指令操作是否产生一个进位（即一个无符号溢出）来设置；V 位根据是否有带符号的溢出来设置。CPSR 中的其他位不受影响。
- 如果<Rd>是程序计数器 R15，则当前程序状态的 SPSR 拷贝到 CPSR。如果处理器处于用户模式或系统模式，则指令的执行结果不可预知。因为这两种模式没有自己的私有 SPSR 寄存器。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd=Rn + shifter_operand+C Flag
    If  S==1  and  Rd==r15  then
        CPSR=SPSR
    Else  if  S==1  then
        N flag=Rd[31]
        Z flag=if  Rd==0  then  1  else  0
        C flag=CarryFrom{ Rn + shifter_operand +C Flag}
        V flag=OverflowFrom{ Rn + shifter_operand +C Flag }
```

4．指令举例

【例 6.6】ADC 指令举例。

ADC 将把两个操作数加起来，并把结果放置到目的寄存器中。它使用一个进位标志位，这样就可以做比 32 位大的加法。下面的例子将加两个 128 位的数。

128 位结果：寄存器 R0、R1、R2 和 R3

第一个 128 位数：寄存器 R4、R5、R6 和 R7

第二个 128 位数：寄存器 8、9、10 和 11。

```
  ADDS    R0, R4, R8          ;加低端的字
  ADCS    R1, R5, R9          ;加下一个字，带进位
```

```
ADCS    R2, R6, R10          ;加第三个字，带进位
ADCS    R3, R7, R11          ;加高端的字，带进位
```

6.7 SBC 带进位的减法指令

1．指令的编码格式

SBC（Subtract with Carry）带进位的减操作指令，用于执行操作数大于 32 位时的减法操作。该指令从寄存器<Rn>中减去<shifter_operand>表示的数值，再减去寄存器 CPSR 中 C 条件标志位的反码（NOT（Carry flag））。并将结果保存到目标寄存器<Rd>中，并根据指令的执行结果设置 CPSR 中相应的标志位。

指令的编码格式如图 6.8 所示。

cond	00	I	0110	S	Rn	Rd	shifter_operand

图 6.8　SBC 指令的编码格式

2．指令的语法格式

```
SBC{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

S 位（bit[20]）决定指令的操作是否影响 CPSR 中条件标志位的值。当 S=1 时指令更新 CPSR 中条件标志位的值；当 S=0 时指令不更新 CPSR 中条件标志位的值。当 S=1 时，有下面两种情况。

- 如果<Rd>不是 R15，CPSR 中的 N 和 Z 位根据指令的执行结果设置。C 位根据指令操作是否产生一个进位（即一个无符号溢出）来设置；V 位根据是否有带符号的溢出来设置。CPSR 中的其他位不受影响。
- 如果<Rd>是程序计数器 R15，则当前程序状态的 SPSR 拷贝到 CPSR。如果处理器处于用户模式或系统模式，则指令的执行结果不可预知。因为这两种模式没有自己的私有 SPSR 寄存器。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）

位和 shifter_operand（bits[11：0]）位。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd=Rn – shifter_operand－NOT（C Flag）
    If  S==1  and  Rd==r15  then
        CPSR=SPSR
    Else  if  S==1  then
        N flag=Rd[31]
        Z flag=if  Rd==0  then  1  else  0
        C flag=NOT BorrowFrom{ Rn - shifter_operand－NOT（C Flag）}
        V flag=OverflowFrom{ Rn - shifter_operand－NOT（C Flag）}
```

4．指令举例

【例 6.7】SBC 指令举例。

下面的程序使用 SBC 实现 64 位减法，（R1，R0）-（R3，R2），结果放到（R1，R0）

```
SUBS    R0，R0，R2;
SBCS    R1，R1，R3;
```

6.8 RSC 带进位的翻转减指令

1．指令的编码格式

RSC（Reverse Subtract with Carry）带进位的翻转减操作指令，从寄存器<shifter_operand>中减去<Rn>表示的数值，再减去寄存器 CPSR 中 C 条件标志位的反码（NOT（Carry Flag）），并将结果保存到目标寄存器<Rd>中，并根据指令的执行结果设置 CPSR 中相应的标志位。

指令的编码格式如图 6.9 所示。

cond	00	I	0111	S	Rn	Rd	shifter_operand

图 6.9　RSC 指令的编码格式

2．指令的语法格式

```
RSC{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

详见 SUB 指令。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

注意 当指令的编码格式中 I 位等于 0，并且移位操作数 shifter_operand 中 bit[7]和 bit[4]也都等于 1，则指令并非 RSC 指令。详情请参阅 ARM 系统结构参考手册。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd= shifter_operand－Rn－NOT{C Flag}
    If  S==1  and  Rd==r15  then
    CPSR=SPSR
    Else  if  S==1  then
        N flag=Rd[31]
        Z flag=if  Rd==0  then  1  else  0
        C flag=NOT BorrowFrom{shifter_operand - Rn －NOT{C Flag}}
        V flag=OverflowFrom{shifter_operand - Rn －NOT{C Flag}}
```

4．指令举例

【例 6.8】RSC 指令举例。

下面程序使用 RSC 指令实现求 64 位数值的负数。

```
RSBS    R2，R0，＃0;
RSC     R3，R1，＃0;
```

6.9　TST 测试指令

1．指令的编码格式

TST（Test）测试指令用于将一个寄存器的值和一个算术值做比较。条件标志位根据两个

操作数做“逻辑与”后的结果设置。

指令的编码格式如图 6.10 所示。

cond	00	I	1000	1	Rn	SBZ	shifter_operand

图 6.10　TST 指令编码格式

2．指令的语法格式

```
TST{<cond>}  <Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rn>

指定第一个源操作数寄存器。

③ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

注意　当指令的编码格式中 I 位等于 0 并且移位操作数 shifter_operand 中 bit[7]和 bit[4]也都等于 1 时，指令并非 TST 指令。详情请参阅 ARM 系统结构参考手册。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    ALU_out = Rn AND shifter_operand
    N Flag = ALU_out[31]
    Z Flag = if ALU_out==0 then 1 else 0
    C Flag =shifter_carry_out
    V Falg=unaffected
```

4．指令举例

【例 6.9】TST 指令举例。

TST 类似于 CMP，不产生放置到目的寄存器中的结果。而是在给出的两个操作数上进行操作并把结果反映到状态标志上。使用 TST 来检查是否设置了特定的位。操作数 1 是要测试的数据字而操作数 2 是一个位掩码。经过测试后，如果匹配则设置 Zero 标志，否则清除它。和 CMP 指令一样，该指令不需要指定 S 后缀。

下面的指令测试在 R0 中是否设置了位 0

```
  TST     R0, #%1
```

6.10 TEQ 测试相等指令

1．指令的编码格式

TEQ（Test Equivalence）测试指令用于将一个寄存器的值和一个算术值做比较。条件标志位根据两个操作数做“逻辑或”后的结果设置。以便后面的指令根据相应的条件标志来判断是否执行。

指令的编码格式如图 6.11 所示。

cond	00	I	1001	1	Rn	SBZ	shifter_operand

图 6.11 TEQ 指令编码格式

注意 TEQ 类似于 TST。区别是这里的概念上的计算是 EOR 而不是 AND。这提供了一种查看两个操作数是否相同而又不影响进位标志（不像 CMP 那样）的方法。

2．指令的语法格式

```
TEQ{<cond>}  <Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rn>

指定第一个源操作数寄存器。

③ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

注意 当指令的编码格式中 I 位等于 0，并且移位操作数 shifter_operand 中 bit[7]和 bit[4]也都等于 1，则指令并非 TEQ 指令。详情请参阅 ARM 系统结构参考手册。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    ALU_out = Rn EOR shifter_operand
    N Flag = ALU_out[31]
    Z Flag = if ALU_out==0 then 1 else 0
    C Flag =shifter_carry_out
    V Falg=unaffected
```

4．指令举例

【例 6.10】TEQ 指令举例。

下面的指令比较 R0 和 R1 是否相等，它该指令不影响 CPSR 中的 V 位和 C 位。

```
TEQ     R0,R1;
```

TST 指令与 EORS 指令的区别在于 TST 指令不保存运算结果。使用 TEQ 进行相等测试，常与 EQ 和 NE 条件码配合使用，当两个数据相等时，条件码 EQ 有效，否则条件码 NE 有效。

6.11 CMP 比较指令

1．指令的编码格式

CMP（Compare）比较指令使用寄存器 Rn 的值减去 operand2 的值，根据操作的结果更新 CPSR 中相应的条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。

指令的编码格式如图 6.12 所示。

cond	00	I	1010	1	Rn	SBZ	shifter_operand

图 6.12 CMP 指令编码格式

2．指令的语法格式

```
CMP{<cond>}  <Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rn>

指定第一个源操作数寄存器。

③ <shifter_operand>

详见 TST 指令。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    ALU_out = Rn – shifter_operand
    N Flag = ALU_out[31]
    Z Flag = if ALU_out==0 then 1 else 0
    C Flag =NOT BorrowFrom{Rn－shifter_operand}
```

```
V Falg=OverflowFrom{Rn－shifter_operand}
```

4．指令举例

【例 6.11】CMP 指令举例。

CMP 允许把一个寄存器的内容与另一个寄存器的内容或立即值进行比较，更改状态标志来允许进行条件执行。它进行一次减法，但不存储结果，而是正确地更改标志位。标志位表示的是操作数 1 与操作数 2 比较的结果（其可能为大、小、相等）。如果操作数 1 大于操作操作数 2，则此后的有 GT 后缀的指令将可以执行。

显然，CMP 不需要显式的指定 S 后缀来更改状态标志。

（1）下面的指令比较 R1 和立即数 10 并设置相关的标志位。

```
CMP     R1,#10
```

（2）下面指令比较寄存器 R1 和 R2 中的值并设置相关的标志位。

```
CMP     R1,R2
```

通过上面的例子可以看出，CMP 指令与 SUBS 指令的区别在于 CMP 指令不保存运算结果，在进行两个数据大小判断时，常用 CMP 指令及相应的条件码来操作。

6.12　CMN 负数比较指令

1．指令的编码格式

CMN（Compare Negative）比较指令使用寄存器 Rn 的值减去 operand2 的负数值（加上 operand2），根据操作的结果更新 CPSR 中相应的条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。

指令的编码格式如图 6.13 所示。

cond	00	I	1011	1	Rn	SBZ	shifter_operand

图 6.13　CMN 指令编码格式

2．指令的语法格式

```
CMN{<cond>}  <Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rn>

指定第一个源操作数寄存器。

③ <shifter_operand>

详见 TST 指令。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    ALU_out = Rn + shifter_operand
    N Flag = ALU_out[31]
    Z Flag = if ALU_out==0 then 1 else 0
    C Flag =CarryFromFrom{Rn+shifter_operand}
    V Falg=OverflowFrom{Rn+shifter_operand}
```

4．指令举例

【例 6.12】CMN 指令举例。

CMN 指令将寄存器<Rn>中的值加上<shifter_operand>表示的数值，根据加法的结果设置 CPSR 中相应的条件标志位。寄存器<Rn>中的值加上<shifter_operand>的操作结果对 CPSR 中条件标志位的影响，与寄存器<Rn>中的值减去<shifter_operand>的操作结果的相反数对 CPSR 中条件标志位的影响有细微差别。当第二个操作数为 0 或者为 0x80000000 时二者结果不同。比如下面两条指令。

```
CMP     Rn，#0;
CMN     Rn，#0;
```

第一条指令使标志位 C 值为 1，第二条指令使标志位 C 值为 0。

下面的指令使 R0 值加 1，判断 R0 是否为 1 的补码，若是，则 Z 置位。

```
CMN     R0，#1;
```

6.13 ORR 逻辑或指令

1．指令的编码格式

ORR（Logical OR）为逻辑或操作指令，将第二个源操作数<shifter_operand>的值与寄存器 Rn 的值按位做逻辑或操作，结果保存到 Rd 中。

指令的编码格式如图 6.14 所示。

cond	00	I	1100	S	Rn	SBZ	shifter_operand

图 6.14 ORR 指令编码格式

2．指令的语法格式

```
ORR{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

S 位（bit[20]）决定指令的操作是否影响 CPSR 中条件标志位的值。当 S=1 时指令更新 CPSR 中条件标志位的值；当 S=0 时指令不更新 CPSR 中条件标志位的值。当 S=1 时，有下面两种情况。

- 如果<Rd>不是 R15，则 CPSR 中的 N 位和 Z 位根据指令的执行结果设置。C 位则根据指令操作是否产生一个进位（即一个无符号溢出）来设置；V 位则根据是否有带符号的溢出来设置。CPSR 中的其他位不受影响。
- 如果<Rd>是程序计数器 R15，则当前程序状态的 SPSR 拷贝到 CPSR。如果处理器处于用户模式或系统模式，则指令的执行结果不可预知。因为这两种模式没有自己的私有 SPSR 寄存器。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd=Rn OR shifter_operand
    If  S==1  and  Rd==r15  then
        CPSR=SPSR
    Else  if  S==1  then
        N flag=Rd[31]
        Z flag=if  Rd==0  then  1  else  0
        C flag=shifter_Carry_out
        V flag=unaffected
```

4．指令举例

【例 6.13】ORR 指令举例。

（1）设置 R0 中位 0 和 1

```
ORR     R0, R0, #3
```

（2）将 R0 的低 4 位置 1

```
ORR     R0,R0,#0x0F;
```

（3）使用 ORR 指令将 R2 的高 8 位数据移入到 R3 的低 8 位中

```
MOV     R1，R2，LSR ＃4;
ORR     R3，R1，R3，LSL ＃8;
```

6.14 BIC 位清零指令

1．指令的编码格式

BIC（Bit Clear）位清零指令，将寄存器 Rn 的值与第二源操作数<shifter_operand>的值的反码按位做“逻辑与”操作，结果保存到 Rd 中。

指令的编码格式如图 6.15 所示。

cond	00	I	1110	S	Rn	SBZ	shifter_operand

图 6.15 BIC 指令编码格式

2．指令的语法格式

```
BIC{<cond>}{S}  <Rn>,<Rn>,<shifter_operand>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <S>

详见 ORR 指令。

③ <Rd>

指定目标寄存器。

④ <Rn>

指定第一个源操作数寄存器。

⑤ <shifter_operand>

使用 ARM 的通用寻址模式确定第二个源操作数。它影响指令编码格式中的 I（bit[25]）位和 shifter_operand（bits[11：0]）位。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd=Rn AND NOT shifter_operand
    If  S==1  and  Rd==r15  then
        CPSR=SPSR
    Else  if  S==1  then
        N flag=Rd[31]
        Z flag=if  Rd==0  then  1  else  0
        C flag=shifter_Carry_out
        V flag=unaffected
```

4．指令举例

【例 6.14】BIC 指令举例。

（1）BIC 指令真值表如表 6.4 所示。

表 6.4　　BIC 指令真值表

Op_1	Op_2	结　果
0	0	0
0	1	0
1	0	1
1	1	0

（2）清除 R0 中的位 0、1 和 3。保持其余的不变。

```
BIC     R0, R0, #0x1011;
```

（3）将 R3 的反码和 R2 逻辑与，结果保存到 R1 中。

```
BIC    R1，R2，R3;
```

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 7 章 乘法指令

华清远见

ARM 乘法指令完成两个数据的乘法。两个 32 位二进制数相乘的结果是 64 位的积。在有些 ARM 的处理器版本中，将乘积的结果保存到两个独立的寄存器中。另外一些版本只将最低有效 32 位存放到一个寄存器中。

无论是哪种版本的处理器，都有乘-累加的变型指令，将乘积连续累加得到总和。而且有符号数和无符号数都能使用。对于有符号数和无符号数，结果的最低有效位是一样的。因此，对于只保留 32 位结果的乘法指令，不需要区分有符号数和无符号数两种情况。

乘法指令的二进制编码格式如图 7.1 所示。

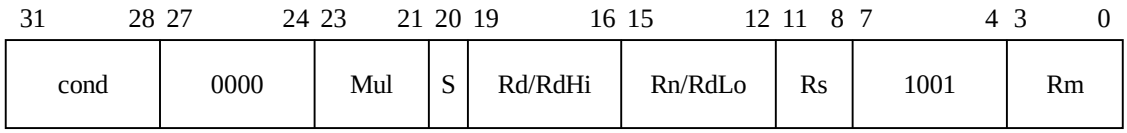


图 7.1　乘法指令的二进制编码

表 7.1 显示了各种形式乘法指令的功能。

表 7.1　　各种形式乘法指令

操作码[23:21]	助记符	意　义	操　作
000	MUL	乘（保留 32 位结果）	Rd：=（Rm×Rs）[31：0]
001	MLA	乘-累加（32 位结果）	Rd：=（Rm×Rs＋Rn）[31：0]
100	UMULL	无符号数长乘	RdHi：RdLo：=Rm×Rs
101	UMLAL	无符号长乘-累加	RdHi：RdLo：＋=Rm×Rs
110	SMULL	有符号数长乘	RdHi：RdLo：=Rm×Rs
111	SMLAL	有符号数长乘-累加	RdHi：RdLo：＋=Rm×Rs

其中：

① “RdHi：RdLo”是由 RdHi（最高有效 32 位）和 RdLo（最低有效 32 位）链接形成的 64 位数，“[31：0]”只选取结果的最低有效 32 位。

② 简单的赋值由“：=”表示。

③ 累加（将右边加到左边）是由“＋=”表示。

同其他数据处理指令一样，位 S 控制条件码的设置。当在指令中设置了位 S 时，则有以下结果。

① 对于产生 32 位结果的指令形式，将标志位 N 设置为 Rd 的第 31 位的值；对于产生长结果的指令形式，将其设置为 RdHi 的第 31 位的值。

② 对于产生 32 位结果的指令形式，如果 Rd 等于零，则标志位 Z 置位；对于产生长结果的指令形式，RdHi 和 RdLo 同时为零时，标志位 Z 置位。

③ 将标志位 C 设置成无意义的值。

④ 标志位 V 不变。

注意　乘法指令不能对第二操作数使用立即数或被移位的寄存器。

7.1 MUL 乘法指令

1．指令编码格式

MUL（Multiply）32 位乘法指令将 Rm 和 Rs 中的值相乘，结果的最低 32 位保存到 Rd 中。

指令的编码格式如图 7.2 所示。

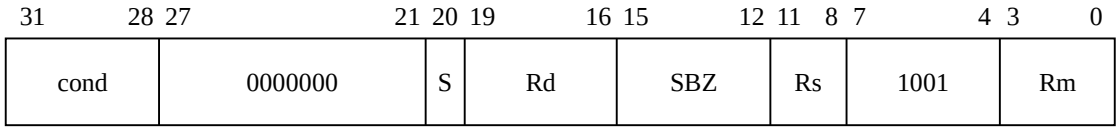


图 7.2 MUL 指令的编码格式

2．指令的语法格式

```
MUL{<cond>}{S}   <Rd>，<Rm>，<Rs>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② S

S 位（bit[20]）决定指令的操作是否影响 CPSR 中的条件标志位 N 位和 Z 位的值。当 S=1 时，更新 CPSR 中的条件标志位的值；当 S=0 时，指令不更新 CPSR 中的条件标志位。

③ <Rd>

寄存器位目标寄存器。

④ <Rm>

第一个乘数所在寄存器。

⑤ <Rs>

第二乘数所在寄存器。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd={Rm*Rs}[31:0]
    If  S==1  then
        N  flag = Rd[31]
        Z  flag = if  Rd==0  then  1  else  0
        C  flag = unaffected
        V  flag = unaffected
```

注意 当程序计数器 r15 被用作<Rd>、<Rm>、<Rs>时，指令的执行结果不可预知；当目的寄存器<Rd>和<Rm>一样时，指令的执行结果不可预知；在 ARM 版本 v5 以后的体系中，在 MULS 指令执行结束后，标志位 C 保持不变，在 v5 以前的版本中，MULS 指令执行后，标志位 C 结果不可预知。

4．指令举例

（1）R1=R2×R3

```
MUL    R1, R2, R3
```

（2）R0=R3×R7，同时设置 CPSR 中 N 位和 Z 位。

```
MULS    R0, R3, R7
```

7.2 MLA 乘-累加指令

1．指令编码格式

MLA（Multiply Accumulate）32 位乘累加指令将 Rm 和 Rs 中的值相乘，再将乘积加上第 3 个操作数，结果的最低 32 位保存到 Rd 中。

指令的编码格式如图 7.3 所示。

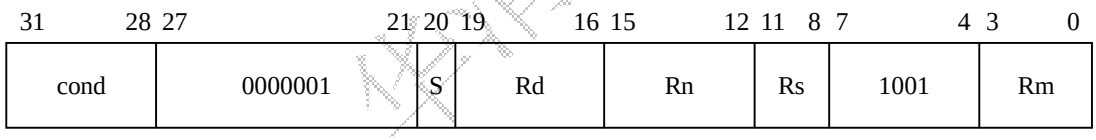


图 7.3 MLA 指令的编码格式

2．指令的语法格式

```
MLA{<cond>}{S}   <Rd>，<Rm>，<Rs>，<Rn>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② S

S 位（bit[20]）决定指令的操作是否影响 CPSR 中的条件标志位 N 位和 Z 位的值。当 S=1 时，更新 CPSR 中的条件标志位的值；当 S=0 时，指令不更新 CPSR 中的条件标志位。

③ <Rd>

寄存器位目标寄存器。

④ <Rm>

第一个乘数所在寄存器。

⑤ <Rs>

第二乘数所在寄存器。

⑥ <Rn>

将要累加到<Rm>×<Rs>结果中的第 3 操作数。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Rd={Rm*Rs + Rn}[31:0]
    If  S==1  then
        N  flag = Rd[31]
        Z  flag = if  Rd==0  then  1  else  0
        C  flag = unaffected
        V  flag = unaffected
```

4．指令举例

下面指令完成 R1=R2×R3＋10 的操作。

```
MOV     R0, #0x0A;
MLA     R1, R2, R3, R0;
```

7.3　UMULL 无符号数长乘指令

1．指令编码格式

UMULL（Unsigned Multiply Long）为 64 位无符号乘法指令。指令将 Rm 和 Rs 中的值做无符号数相乘，结果的低 32 位保存到 RsLo 中，而高 32 位保存到 RdHi 中。

指令的编码格式如图 7.4 所示。

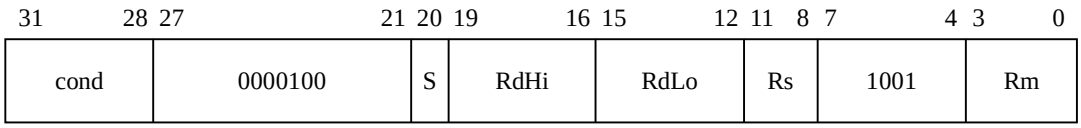


图 7.4　UMULL 指令的编码格式

2．指令的语法格式

```
UMULL{<cond>}{S}   <RdLo>，<RdHi>，<Rm>，<Rs>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② S

S 位（bit[20]）决定指令的操作是否影响 CPSR 中的条件标志位 N 位和 Z 位的值。当 S=1 时，更新 CPSR 中的条件标志位的值；当 S=0 时，指令不更新 CPSR 中的条件标志位。

③ <RdLo>

寄存器位目标寄存器。存储结果的低 32 位值。

④ <RdHi>

寄存器位目标寄存器。存储结果的高 32 位值。

⑤ <Rm>

第一乘数寄存器。

⑥ <Rn>

第二乘数寄存器。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    RdHi={Rm*Rs }[63:32]
    RdLo={Rm*Rs }[31:0]
    If  S==1  then
        N  flag = RdHi[31]
        Z  flag = if  （（RdHi==0）and（RdLo==0））then  1  else  0
        C  flag = unaffected
        V  flag = unaffected
```

4．指令举例

下面指令完成（R1，R0）=R5×R8 操作。

```
UMULL    R0, R1, R5, R8;
```

7.4 UMLAL 无符号长乘-累加操作指令

1．指令编码格式

UMLAL（Unsigned Multiply Accumulate Long）为 64 位无符号长乘-累加指令。指令将 Rm 和 Rs 中的值做无符号数相乘，64 位乘积与 RdHi，RdLo 相加，结果的低 32 位保存到 RsLo 中，而高 32 位保存到 RdHi 中。

指令的编码格式如图 7.5 所示。

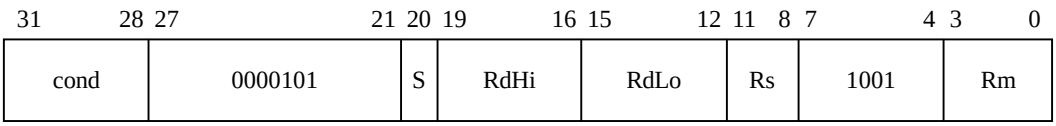


图 7.5　UMALL 指令的编码格式

2．指令的语法格式

```
UMALL{<cond>}{S}   <RdLo>，<RdHi>，<Rm>，<Rs>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② S

S 位（bit[20]）决定指令的操作是否影响 CPSR 中的条件标志位 N 位和 Z 位的值。当 S=1 时，更新 CPSR 中的条件标志位 N 位和 Z 位的值；当 S=0 时，指令不更新 CPSR 中的条件标志位。

③ <RdLo>

存储将要累加到<Rm>×<Rn>乘积结果中的加数的低 32 位数值的寄存器；同时也为寄存器位目标寄存器，存储最终结果的低 32 位值。

④ <RdHi>

存储将要累加到<Rm>×<Rn>乘积结果中的加数的高 32 位数值的寄存器；同时也为寄存器位目标寄存器，存储最终结果的高 32 位值。

⑤ <Rm>

第一乘数寄存器。

⑥ <Rn>

第二乘数寄存器。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    RdHi={Rm*Rs }[63:32]+RdHi+CarryFrom{{Rm*Rs}[31:0]+RdLo}
    RdLo={Rm*Rs }[31:0]+RdLo
    If  S==1  then
        N  flag = RdHi[31]
        Z  flag = if  ((RdHi==0) and (RdLo==0)) then  1  else  0
        C  flag = unaffected
        V  flag = unaffected
```

4．指令举例

下面的指令完成（R1，R0）= R5×R8＋（R1，R0）操作。

```
UMLAL   R0, R1, R5,R8;
```

7.5 SMULL 无符号长乘-累加操作指令

1．指令编码格式

SMULL（Signed Multiply Long）64 位有符号长乘法指令。指令将 Rm 和 Rs 中的值做有符号数相乘，结果的低 32 位保存到 RsLo 中，而高 32 位保存到 RdHi 中。

指令的编码格式如图 7.6 所示。

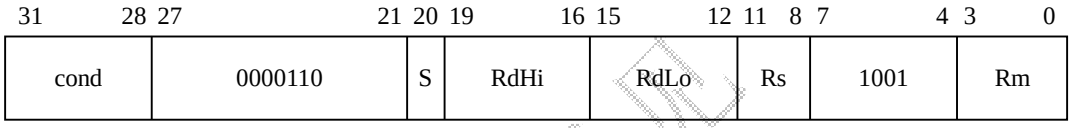


图 7.6 SMULL 指令的编码格式

2．指令的语法格式

```
SMULL{<cond>}{S}   <RdLo>，<RdHi>，<Rm>，<Rs>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② S

S 位（bit[20]）决定指令的操作是否影响 CPSR 中的条件标志位 N 位和 Z 位的值。当 S=1 时，更新 CPSR 中的条件标志位 N 位和 Z 位的值；当 S=0 时，指令不更新 CPSR 中的条件标志位。

③ <RdLo>

寄存器位目标寄存器，存储最终结果的低 32 位值。

④ <RdHi>

寄存器位目标寄存器，存储最终结果的高 32 位值。

⑤ <Rm>

第一乘数寄存器。

⑥ <Rn>

第二乘数寄存器。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    RdHi={Rm*Rs }[63:32]
    RdLo={Rm*Rs }[31:0]
    If  S==1  then
        N  flag = RdHi[31]
        Z  flag = if  （（RdHi==0）and（RdLo==0））then  1  else  0
        C  flag = unaffected
        V  flag = unaffected
```

4．指令举例

下面的指令完成（R3，R2）= R7×R6 操作。

```
SMULL    R2, R3, R7,R6;
```

7.6　SMLAL 有符号长乘-累加操作指令

1．指令编码格式

SMLAL（Signed Multiply Accumulate Long）为 64 位有符号长乘法指令。指令将 Rm 和 Rs 中的值做有符号数相乘，64 位乘积与 RdHi，RdLo 相加，结果的低 32 位保存到 RsLo 中，而高 32 位保存到 RdHi 中。

指令的编码格式如图 7.7 所示。

31 28	27 21	20	19 16	15 12	11 8	7 4	3 0
cond	0000111	S	RdHi	RdLo	Rs	1001	Rm

图 7.7　SMLAL 指令的编码格式

2．指令的语法格式

```
SMLAL{<cond>}{S}   <RdLo>，<RdHi>，<Rm>，<Rs>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② S

S 位（bit[20]）决定指令的操作是否影响 CPSR 中的条件标志位 N 位和 Z 位的值。当 S=1 时，更新 CPSR 中的条件标志位 N 位 Z 位的值；当 S=0 时，指令不更新 CPSR 中的条件标志位。

其他参数详见 SMULL 指令。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    RdHi={Rm*Rs }[63:32]+RdHi+CarryFrom{{Rm*Rs}[31:0]+RdLo}
    RdLo={Rm*Rs }[31:0]+RdLo
    If  S==1  then
        N  flag = RdHi[31]
        Z  flag = if  ((RdHi==0) and (RdLo==0)) then  1  else  0
        C  flag = unaffected
        V  flag = unaffected
```

4．指令举例

下面的指令完成（R3，R2）= R7×R6+（R3，R2）操作。

```
SMLAL    R2, R3, R7,R6;
```

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 8 章 跳转指令

华清远见

跳转（B）和跳转连接（BL）指令是改变指令执行顺序的标准方式。ARM 一般按照字地址顺序执行指令，需要时使用条件执行跳过某段指令。只要程序必须偏离顺序执行，就要使用控制流指令来修改程序计数器。尽管在特定情况下还有其他几种方式实现这个目的，但转移和转移连接指令是标准的方式。

跳转指令改变程序的执行流程或者调用子程序。这种指令使得一个程序可以使用子程序、if-then-else 结构以及循环。执行流程的改变迫使程序计数器 PC 指向一个新的地址，ARMv5 架构指令集包含的跳转指令如表 8.1 所示。

表 8.1 ARMv5 架构跳转指令

助记符	说　明	操　作
B	跳转指令	pc←label
BL	带返回的连接跳转	pc←label(lr←BL 后面的第一条指令)
BX	跳转并切换状态	pc←Rm&0xfffffffe, T←Rm&1
BLX	带返回的跳转并切换状态	pc←lable, T←1 pc←Rm&0xfffffffe, T←Rm&1 lr←BL 后面的第一条指令

另一种实现指令跳转的方式是通过直接向 PC 寄存器中写入目标地址值，实现在 4GB 地址空间中任意跳转，这种跳转指令又被称为长跳转。如果在长跳转指令之前使用“MOV　LR”或“MOV　PC”等指令，可以保存将来返回的地址值，也就实现了在 4GB 的地址空间中的子程序调用。

在 ARMv5 以前的版本中，传送到 PC 寄存器中的目标地址值的低两位 bits[1：0]被忽略，跳转指令只能在 ARM 指令集中执行，即程序不能从 ARM 状态切换到 Thumb 状态。在非 T 系列版本 5 的 ARM 体系不含 Thumb 指令，当程序试图切换到 Thumb 状态时，将产生未定义指令异常中断。

在 ARMv5 以后的版本中，有两种类型的带连接的跳转切换指令（BLX），叙述如下。

（1）形式如“BLX　<Rm>”，它是一种类似于带寄存器 Rm 的 BX 指令。指令执行 BX 操作，同时将返回地址放到 LR 寄存器中。这种形式的带状态切换的跳转连接指令，方便了 ARM/Thumb 互交的子程序调用。

（2）另一种类型的 BLX 指令类似于 BL 指令，指令使程序跳转到指定地址，并将返回地址保存到 LR 寄存器中，该指令能够实现 32MB 地址空间的跳转。与 BL 指令的不同之处在于它返回到 Thumb 状态，而不是 ARM 状态。

8.1　跳转指令 B 及带连接的跳转指令 BL

1．指令编码格式

跳转指令 B 使程序跳转到指定的地址执行程序。带连接的跳转指令 BL 将下一条指令的

地址拷贝到 r14（即返回地址连接寄存器 LR）寄存器中，然后跳转到指定地址运行程序。需要注意的是，这两条指令和目标地址处的指令都要属于 ARM 指令集。两条指令都可以根据 CPSR 中的条件标志位的值决定指令是否执行。

指令的编码格式如图 8.1 所示。

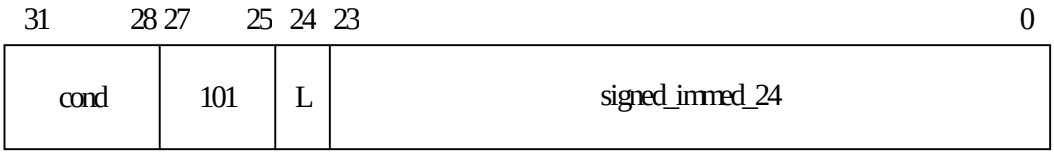


图 8.1　B&BL 指令编码格式

2．指令的语法格式

```
B{L}{<cond>}  <target_address>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② L

L 位（bit[24]）=1，指令存储返回地址到 LR；L 位（bit[24]）=0，指令仅实现跳转，不保存返回指令。

③ <target_addrss>

指令跳转的目标地址。指令通过下面的方法计算目标地址。

- 将 24 位的立即数符号扩展为 32 位。
- 将扩展后的 32 位立即数左移两位。
- 将得到的值加到 PC 寄存器中，即得到跳转的目标地址。

注意　由于以上原因，B 和 BL 指令只能实现 ± 32MB 空间的跳转。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  conditionPassed{cond}  then
    If  L==1  then
        LR = address of the instruction after the branch instruftion
    PC = PC + (SignExtend(signed_immed_24)<<2)
```

4．指令的使用

BL 指令用于实现子程序调用。子程序的返回可以通过将 LR 寄存器的值复制到 PC 寄存器来实现。下面三种指令可以实现子程序返回。

- BX　r14（如果体系结构支持 BX 指令）。
- MOV　PC，r14。

- 当子程序在入口处使用了压栈指令：

```
STMFD  r13!，{<registers>，r14}，
```

可以使用指令。

```
LDMFD  r13!，{<registers>，PC}
```

将子程序返回地址放入 PC 中。

ARM 汇编器通过以下步骤计算指令编码中的 signed_immed_24。

（1）将 PC 寄存器的值作为本跳转指令的基地址值。

（2）从跳转的目标地址中减去上面所说的跳转的基地址，生成字节偏移量。由于 ARM 指令是字对齐的，该字节偏移量为 4 的倍数。

（3）当上面生成的字节偏移量超过－33554432～+33554430 时，不同的汇编器使用不同的代码产生策略。

（4）否则，将指令编码字中的 signed_immed_24 设置成上述字节偏移量的 bits[25：2]。

注意 在一些 RISC 体系结构的处理器中，存在延时跳转（delayed branch）模式，即在程序执行跳转指令跳转到目标地址之前，程序会执行跳转指令之后的指令。但在 ARM 体系中，没有这种延时跳转机制。

5．指令举例

（1）程序跳转到 LABLE 标号处。

```
    B  LABLE ;
    ADD  r1，r2，＃4
    ADD  r3，r2，＃8
    SUB  r3，r3，r1
LABLE
    SUB  r1，r2，#8
```

（2）跳转到绝对地址 0x1234 处。

```
B  0x1234
```

（3）跳转到子程序 func 处执行，同时将当前 PC 值保存到 LR 中。

```
BL  func
```

（4）条件跳转：当 CPSR 寄存器中的 C 条件标志位为 1 时，程序跳转到标号 LABLE 处执行。

```
BCC  LABLE
```

（5）通过跳转指令建立一个无限循环。

```
LOOP
```

```
    ADD  r1，r2，＃4
    ADD  r3，r2，＃8
    SUB  r3，r3，r1
    B  LOOP
```

（6）通过使用跳转使程序体循环 10 次。

```
    MOV  r0，＃10
LOOP
    SUBS  r0，＃1
    BNE   LOOP
```

（7）条件子程序调用示例。

```
……
CMP  r0，＃5                ;如果 r0<5
BLLT  SUB1                  ;则调用
BLGE  SUB2                  ;否则调用 SUB2
```

注意　只有 SUB1 不改变条件码，本例才能正确执行，因为如果 BLLT 执行了转移，将返回到 BLGE 指令。如果条件码被 SUB1 子程序改变，则 SUB2 可能又会被执行，从而达不到指令的预期效果。

8.2　带状态切换的跳转指令 BX

1．指令编码格式

带状态切换的跳转指令 BX 使程序跳转到指令中指定的参数 Rm 指定的地址执行程序，Rm 的第 0 位拷贝到 CPSR 中 T 位，位[31：1]移入 PC。若 Rm 的 bit[0]为 1，则跳转时自动将 CPSR 中的标志位 T 置位，即把目标地址的代码解释为 Thumb 代码；若 Rm 的位 bit[0]为 0，则跳转时自动将 CPSR 中的标志位 T 复位，即把目标地址代码解释为 ARM 代码。

指令的编码格式如图 8.2 所示。

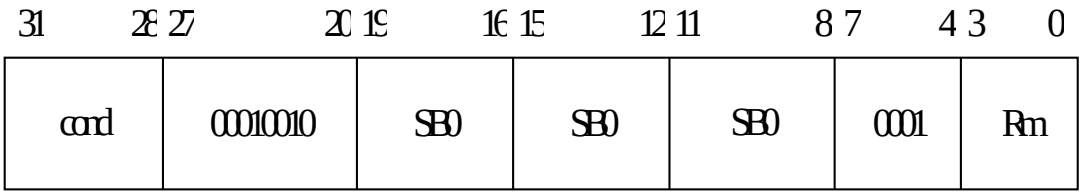


图 8.2　BX 指令编码格式

2．指令的语法格式

```
BX{<cond>}  <Rm>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rm>

包含跳转指令的目标地址。如果 Rm 的 bit[0]=0，目标地址处指令为 ARM 指令；如果 Rm 的 bit[0]=1，目标地址处指令为 Thumb 指令。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  conditionPassed{cond}  then
    T  Flag=Rm[0]
    PC = Rm AND 0xfffffffe
```

4．指令的使用

- 当 Rm[1：0]=0b10 时，指令的执行结果不可预知。因为在 ARM 状态下，指令是 4 字节对齐的。
- PC 可以作为 Rm 寄存器使用，但这种用法不推荐使用。当 PC 作为<Rm>使用时，指令“BX　PC”将程序跳转到当前指令下面第二条指令处执行。虽然这样跳转可以实现，但最好使用下面的指令完成这种跳转。

```
MOV   PC, PC
```

或，

```
ADD   PC, PC, #0
```

5．指令举例

（1）转移到 r0 中的地址，如果 r0[0]=1，则进入 Thumb 状态。

```
BX   r0;
```

（2）跳转到 r0 指定的地址，并根据 r0 的最低位来切换处理器状态。

```
ADRL   r0，ThumbFun+1 ;
BX   r0;
```

8.3 带状态切换的连接跳转指令 BLX（1）

1．指令编码格式

带连接和状态切换的跳转指令 BLX（Branch with Link Exchange）使用标号，用于使程序跳转到 Thumb 状态或从 Thumb 状态返回。该指令为无条件执行指令，并用分支寄存器的最低位来更新 CPSR 中的 T 位，将返回地址写入到连接寄存器 LR 中。

指令编码格式如图 8.3 所示。

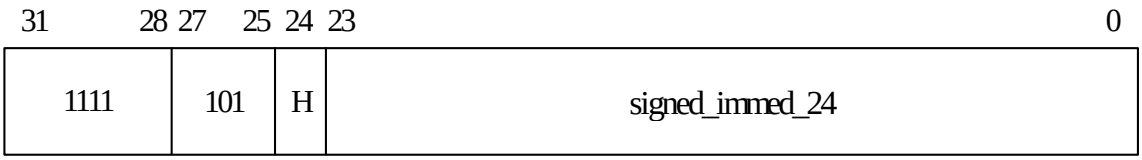


图 8.3　BLX(1)指令编码格式

2．语法格式

```
BLX  <target_add>
```

其中，<target_add>为指令的跳转目标地址。该地址根据以下规则计算。

① 将指令中指定的 24 位偏移量进行符号扩展，形成 32 位立即数。

② 将结果左移两位。

③ 位 H（bit[24]）加到结果地址的第一位（bit[1]）。

④ 将结果累加进程序计数器 PC 中。

计算偏移量的工作一般由 ARM 汇编器来完成。这种形式的跳转指令只能实现±32MB 空间的跳转。

左移两位形成字偏移量，然后将其累加进程序计数器 PC 中。这时，程序计数器的内容为 BX 指令地址加 8 字节。位 H（bit[24]）也加到结果地址的第一位（bit[1]），使目标地址成为半字地址，以执行接下来的 Thumb 指令。计算偏移量的工作一般由 ARM 汇编器来完成。这种形式的跳转指令只能实现±32MB 空间的跳转。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

第一种格式 BLX 指令。

```
LR=address of the instruction after the BLX instruction
T Flag=1
PC=PC + PC = PC + (SignExtend(signed_immed_24)<<2) + (H<<1)
```

4．指令的使用

- 从 Thumb 状态返回到 ARM 状态，使用 BX 指令。

```
BX  r14
```

- 可以在子程序的入口和出口增加栈操作指令。

```
PUSH  {<registers>，r14}
……
POP  {<registers>，PC}
```

8.4 带状态切换的连接跳转指令 BLX（2）

1．指令编码格式

带连接和状态切换的跳转指令 BLX（Branch with Link Exchange）使用一个寄存器中的绝对地址，用于使程序跳转到 Thumb 状态或从 Thumb 状态返回。该指令用分支寄存器的最低位来更新 CPSR 中的 T 位，将返回地址写入到连接寄存器 LR 中。

指令编码格式如图 8.4 所示。

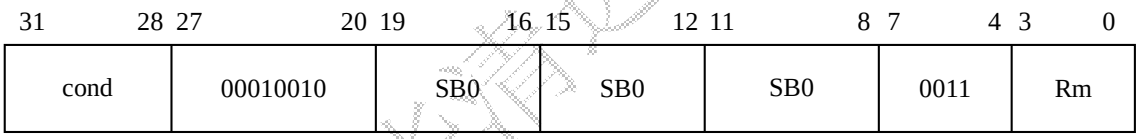


图 8.4 BLX(2)指令编码格式

2．语法格式

```
BLX{<cond>}  <Rm>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rm>

寄存器 Rm 指定转移目标，Rm 的第 0 位拷贝到 CPSR 中的 T 位，bit[31：0]移入 PC。

- 如果 Rm 的 bit[0]=1，则跳转时自动将 CPSR 中的标志位 T 置位，即把目标地址的代码解释为 Thumb 代码。
- 如果 Rm 的 bit[0]=0，则跳转时自动将 CPSR 中的标志位 T 复位，即把目标地址代码解释为 ARM 代码。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPass{cond}  then
      LR = address of the instruction after the branch instruction
      T Flag=Rm[0]
      PC=Rm AND 0xfffffffe
```

注意　在这种情况下，如果 Rm 的 bit[1：0]=0b10，指令的执行结果不可预知，因为这将导致在 ARM 状态下非对齐的字访问。

4．指令举例

调用 Thumb 子程序。

```
    CODE32                  ;ARM 代码
    ……
    BLX     TSUB            ;调用 Thumb 子程序
    ……
    CODE16                  ;Thumb 代码开始
TSUB
    ……
    BX      r14             ;返回 ARM 状态
```

注意　（1）一些不支持 Thumb 指令集的 ARM 处理器将捕获这些指令，允许软件仿真 Thumb 指令。
（2）只有实现 ARMv5 版本以上的处理器支持 BLX 指令的两种格式。

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 9 章 协处理器及其他指令

华清远见

9.1 协处理器指令

ARM 体系结构允许通过增加协处理器来扩展指令集。最常用的协处理器是用于控制片上功能的系统协处理器。例如控制 Cache 和存储管理单元的 CP15 寄存器。此外，还有用于浮点运算的浮点 ARM 协处理器，各生产商还可以根据需要开发自己的专用协处理器。

ARM 协处理器具有自己专用的寄存器组，它们的状态由控制 ARM 状态的指令的镜像指令来控制。

程序的控制流指令由 ARM 处理器来处理，所有协处理器指令只能同数据处理和数据传送有关。按照 RISC 的 Load/Store 体系原则，数据的处理和传送指令是被清楚分开的，所以它们有不同的指令格式。

ARM 处理器支持 16 个协处理器，在程序执行过程中，每个协处理器忽略 ARM 和其他协处理器指令。当一个协处理器硬件不能执行属于它的协处理器指令时，将产生一个未定义指令异常中断，在该异常中断处理过程中，可以通过软件仿真该硬件操作。如果，一个系统中不包含向量浮点运算器，则可以选择浮点运算软件包来支持向量浮点运算。

ARM 协处理器可以部分地执行一条指令，而后产生中断。如除法运算除数为 0 和溢出，这样可以更好地处理运行时产生（run-time-generated）的异常。但是，指令的部分执行是由协处理器完成的，此过程对 ARM 来说是透明的。当 ARM 处理器重新获得执行时，它将从产生异常的指令处开始执行。

对某一个协处理器来说，并不一定用到协处理器指令中的所有的域。具体协处理器如何定义和操作完全由协处理器的制造商自己决定，因此 ARM 协处理器指令中的协处理器寄存器的标识符以及操作助记符也有各种不同的实现定义。程序员可以通过宏定义这些指令的语法格式。

ARM 协处理器指令分以下 3 类。

- 协处理器数据操作。协处理器数据操作完全是协处理器内部操作，它完成协处理器寄存器的状态改变。如浮点加运算，在浮点协处理器中两个寄存器相加，结果放在第 3 个寄存器中。这类指令包括 CDP 指令。
- 协处理器数据传送指令。这类指令从寄存器读取数据装入协处理器寄存器，或将协处理器寄存器的数据装入存储器。因为协处理器可以支持自己的数据类型，所以每个寄存器传送的字数与协处理器有关。ARM 处理器产生存储器地址，但传送的字节由协处理器控制。这类指令包括 LDC 和 STC 指令。
- 协处理器寄存器传送指令。在某些情况下，需要 ARM 处理器和协处理器之间传送数据。如一个浮点运算协处理器，FIX 指令从协处理器寄存器取得浮点数据，将它转换为整数，并将整数传送到 ARM 寄存器中。经常需要用浮点比较产生的结果来影响控制流，因此，比较结果必须传送到 ARM 的 CPSR 中。这类协处理器寄存器传送指令包括 MCR 和 MRC。

表 9.1 列出了所有协处理器处理指令。

表 9.1 协处理器指令

助 记 符	操 作

CDP	协处理器数据操作
LDC	装载协处理器寄存器
MCR	从 ARM 寄存器传数据到协处理器寄存器
MRC	从协处理器寄存器传数据到 ARM 寄存器
STC	存储协处理器寄存器

9.1.1 协处理器数据操作指令 CDP

1．指令编码格式

此指令用于控制数据在协处理器寄存器内部的操作。通常情况下该指令由协处理器完成，如果协处理器不能成功地执行该操作，将产生未定义指令异常。

指令的编码格式如图 9.1 所示。

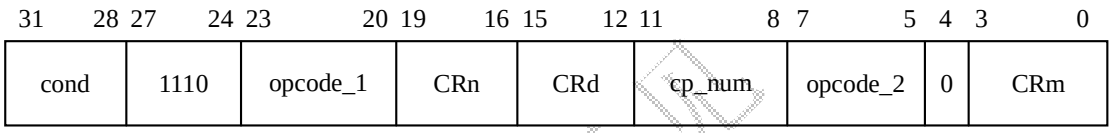


图 9.1 CDP 指令编码格式

2．指令的语法格式

```
CDP{<cond>}  <coproc>，<opcode_1>，<CRd>，<CRn>，<CRm>，<opcode_2>
CDP2  <coproc>，<opcode_1>，<CRd>，<CRn>，<CRm>，<opcode_2>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② CDP2

协处理器数据操作指令 CDP 的一种特殊格式。这种格式中指定编码的条件域<cond>为 ob1111。这种设计为协处理器的设计者提供了一个灵活的扩展空间。此指令只能无条件执行。

③ <coproc>

指定协处理器的编号，标准的协处理器的名字为 p0、p1、…、p15。

④ <opcode_1>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。

⑤ <CRd>

作为目标寄存器的协处理器寄存器。

⑥ <CRn>

确定包含第一个操作数的协处理器寄存器。

⑦ <CRm>

确定包含第二个操作数的协处理器寄存器。

⑧ <opcode_2>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。通常与<opcode_1>配合使用。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Coprocessor[cp_num]－dependent operation
```

注意 CDP 指令通常被用来初始化协处理器。比如在作浮点运算操作时，使用 CDP 指令初始化协处理器寄存器。

4．指令举例

对协处理器 P15 进行操作。第一操作数 opcode_1=2，第二操作数 opcode_2=4，目标寄存器为协处理器寄存器 c12，源寄存器分别为协处理器寄存器 c10 和 c3。

```
CDP   p15，2，c12，c10，c3，4
```

5．指令的使用

- CDP 指令一般用于初始化协处理器，对 ARM 寄存器和存储器没有任何影响。
- 指令的编码格式中，bits[31：24]、bits[11：8]和 bit[4]为 ARM 体系结构定义。其他域由各生产商定义。
- 硬件协处理器支持与否完全由生产商定义，某款 ARM 芯片中，是否支持协处理器或支持哪个协处理器与 ARM 版本无关。生产商可以选择实现部分协处理器指令或者完全不支持协处理器。

9.1.2 协处理器数据读取指令 LDC

1．指令编码格式

LDC（Load Coprocessor）指令通过一定的寻址模式从一系列连续的内存单元将数据读取到协处理器的寄存器中。如果协处理器不能成功地执行操作，将产生未定义的指令异常中断。

指令的编码格式如图 9.2 所示。

31 28	27 25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	110	P	U	N	W	1	Rn	CRd	cp_num	8_bit_word_offset

图 9.2 LDC 指令编码格式

2．指令的语法格式

```
LDC{<cond>}{L}   <coproc>，<CRd>，<addressing_mode>
LDC2{L}          <coproc>，<CRd>，<addressing_mode>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② LDC2

协处理器数据读取指令 LDC 的一种特殊格式。这种格式中指定编码的条件域<cond>为 ob1111。这种设计为协处理器的设计者提供了一个灵活的扩展空间。此指令只能无条件执行。

③ <coproc>

指定协处理器的编号，标准的协处理器的名字为 p0、p1、…、p15。

④ L

长读取操作指示域。设置指令编码格式中的 Nbit（bit[22]），如果该位设置为 1，说明指令是一个长读取指令；该位为 0，说明指令为短读取指令。该指令常用于双精度数据传送。

⑤ <CRd>

确定协处理器目的寄存器。

⑥ <addressing_mode>

确定指令的寻址方式。它将指定指令编码格式中的 P、U、Rn、W 和 8_bit_word_offset 域。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If   ConditionPassed{cond}  then
     Address=start_address
     load  Memory[address,4]  for  Coprocess[cp_num]
     while  {NotFinished{Conprocess[cp_num]}}
           address=address+4
           load Memory[address,4]  for  Coprocessor[cp_num]
     assert  address==end_address
```

4．指令举例

（1）将数据从内存传送到协处理器 p6 寄存器 c1 中，使用寄存器寻址模式，将内存地址放到 ARM 寄存器 r4 中。

```
LDC   p6，CR1，[r4]
```

（2）将数据从内存传送到协处理器 p6 寄存器 c4 中，使用寄存器变址寻址。

```
LDC   p6，CR4，[r2，＃4]
```

5．指令的使用

- 指令的编码格式中，bits[31：23]、bits[21：16]和 bits[11：0]为 ARM 体系结构定义。其他域由各生产商定义。
- 协处理器数据读取指令忽略地址后两位。如果系统中定义了系统控制协处理器，而且地址对齐检测使能打开，当地址 bits[1：0]！=0b00 时，产生地址对齐异常。
- 硬件协处理器支持与否完全由生产商定义，某款 ARM 芯片中，是否支持协处理器或支持哪个协处理器与 ARM 版本无关。生产商可以选择实现部分协处理器指令或者完全不支持协处理器。
- 指令中字的传送数目由协处理器控制。ARM 将连续产生后续地址，直到协处理器指示传送应该结束。在数据传送过程中，ARM 将不影响中断请求，所以协处理器设计者应该注意不应因为传送非常长的数据而损坏系统的中断响应时间。

9.1.3 协处理器数据写入指令 STC

1．指令编码格式

STC（Store Coprocessor）指令通过一定的寻址模式将协处理器寄存器中的数据存储到一系列连续的内存单元中。如果协处理器不能成功地执行操作，将产生未定义的指令异常中断。

指令的编码格式如图 9.3 所示。

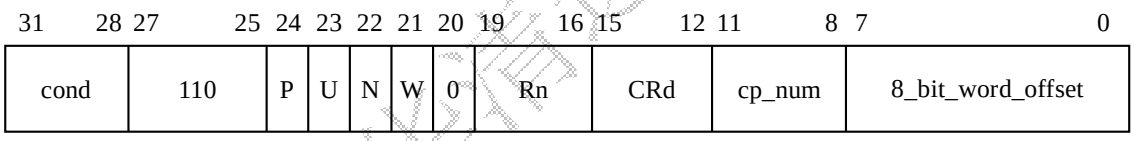


图 9.3 STC 指令编码格式

2．指令的语法格式

```
STC{<cond>}{L}   <coproc>，<CRd>，<addressing_mode>
STC2{L}          <coproc>，<CRd>，<addressing_mode>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② STC2

协处理器数据写入指令 STC 的一种特殊格式。这种格式中指定编码的条件域<cond>为 ob1111。这种设计为协处理器的设计者提供了一个灵活的扩展空间。此指令只能无条件执行。

③ <coproc>

指定协处理器的编号，标准的协处理器的名字为 p0、p1、…、p15。

④ L

长写入操作指示域。设置指令编码格式中的 Nbit（bit[22]），如果该位设置为 1，说

明指令是一个长写入指令；该位为 0，说明指令为短写入指令。该指令常用于双精度数据传送。

⑤ <CRd>

确定协处理器目的寄存器。

⑥ <addressing_mode>

确定指令的寻址方式。它将指定指令编码格式中的 P、U、Rn、W 和 8_bit_word_offset 域。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If   ConditionPassed{cond}  then
     Address=start_address
     Memory[address,4] = value from Coprocess[cp_num]
     while  {NotFinished{Conprocess[cp_num]}}
           address=address+4
           Memory[address,4] = value from Coprocessor[cp_num]
     assert  address==end_address
```

4．指令举例

（1）将协处理器 p8 和寄存器 c8 的数据写入存储器中。寻址模式采用后寄存器寻址变址模式，内存基地址放入 ARM 寄存器 r2 中。

```
STC   p8，CR8，[r2，＃4]!
```

（2）将协处理器 p8 和寄存器 c9 的数据写入存储器中。

```
STC   p8，CR9，[r2]，＃-16
```

5．指令的使用

详见 LDC 指令。

9.1.4　ARM 寄存器到协处理器寄存器的数据传送指令 MCR

1．指令编码格式

ARM 寄存器到协处理器寄存器的数据传送指令 MCR（Move to Coprocessor from ARM Register）将 ARM 寄存器<Rd>的值传送到协处理器寄存器 cp_num 中。如果没有协处理器执行指定操作，将产生未定义指令异常。

指令的编码格式如图 9.4 所示。

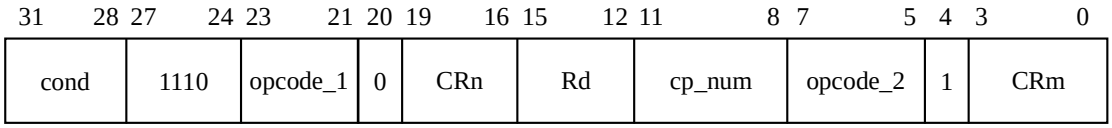


图 9.4　MCR 指令编码格式

2．指令的语法格式

```
MCR{<cond>}  <coproc>，<opcode_1>，<Rd>，<CRn>，<CRm>{，<opcode_2>}
MCR2         <coproc>，<opcode_1>，<Rd>，<CRn>，<CRm>{，<opcode_2>}
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② MCR2

MCR2 指令的一种特殊格式。这种格式中指定编码的条件域<cond>为 ob1111。这种设计为协处理器的设计者提供了一个灵活的扩展空间。此指令只能无条件执行。

③ <coproc>

指定协处理器的编号，标准的协处理器的名字为 p0、p1、…、p15。

④ <opcode_1>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。

⑤ <Rd>

确定哪一个 ARM 寄存器的数值将被传送。如果程序计数器 PC 的值被传送，指令的执行结果不可预知。

⑥ <CRn>

确定包含第一个操作数的协处理器寄存器。

⑦ <CRm>

确定包含第二个操作数的协处理器寄存器。

⑧ <opcode_2>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。通常与<opcode_1>配合使用。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Send  Rd value to coprocessor[cp_num]
```

4．指令举例

将 ARM 寄存器 r7 中的值传送到协处理器 p14 的寄存器 c7 中，第一操作数 opcode_1=1，

第二操作数 opcode_2=6。

```
MCR   p14，1，r7，c7，c12，6
```

5．指令的使用

- 指令的编码格式中，bits[31：24]、bit[20]、bits[15：8]和 bit[4]为 ARM 体系结构定义。其他域由各生产商定义。
- 硬件协处理器支持与否完全由生产商定义，某款 ARM 芯片中，是否支持协处理器或支持哪个协处理器与 ARM 版本无关。生产商可以选择实现部分协处理器指令或者完全不支持协处理器。

9.1.5　协处理器寄存器到 ARM 寄存器的数据传送指令 MRC

1．指令编码格式

协处理器寄存器到 ARM 寄存器的数据传送指令 MRC（Move to ARM register from Coprocessor）将协处理器 cp_num 的寄存器的值传送到 ARM 寄存器中。如果没有协处理器执行指定操作，将产生未定义指令异常。

指令的编码格式如图 9.5 所示。

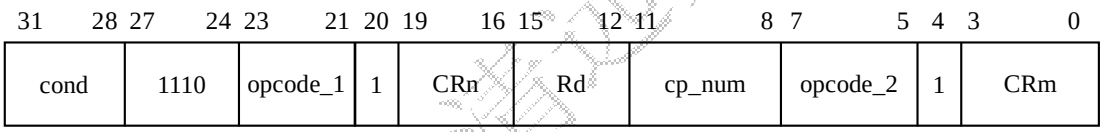


图 9.5　MRC 指令编码格式

2．指令的语法格式

```
MRC{<cond>}  <coproc>，<opcode_1>，<Rd>，<CRn>，<CRm>{，<opcode_2>}
MRC2         <coproc>，<opcode_1>，<Rd>，<CRn>，<CRm>{，<opcode_2>}
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② MRC2

MRC2 指令的一种特殊格式。这种格式中指定编码的条件域<cond>为 ob1111。这种设计为协处理器的设计者提供了一个灵活的扩展空间。此指令只能无条件执行。

③ <coproc>

指定协处理器的编号，标准的协处理器的名字为 p0、p1、…、p15。

④ <opcode_1>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。

⑤ <Rd>

确定哪一个 ARM 寄存器接受协处理器传送的数值。如果程序计数器 PC 被用作目的寄

存器，指令的执行结果不可预知。

⑥ <CRn>

确定包含第一个操作数的协处理器寄存器。

⑦ <CRm>

确定包含第二个操作数的协处理器寄存器。

⑧ <opcode_2>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。通常与<opcode_1>配合使用。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Data=value from coprocessor[cp_num]
    If Rd is R15 then
         N  flag = data[31]
         Z  flag = data[30]
         C  flag = data[29]
         V  flag = data[28]
    Else  /*Rd ≠R15*/
         Rd = data
```

4．指令举例

协处理器源寄存器为 c0 和 c2，目的寄存器为 ARM 寄存器 r4，第一操作数 opcode_1=5，第二操作数 opcode_2=3。

```
MRC  p15，5，r4，c0，c2，3
```

5．指令的使用

- 如果目的寄存器为程序计数器 r15，则程序状态字条件标准位根据传送数据的前 4bit 确定，后 28bit 被忽略。
- 指令的编码格式中，bits[31：24]、bit[20]、bits[15：8]和 bit[4]为 ARM 体系结构定义。其他域由各生产商定义。
- 硬件协处理器支持与否完全由生产商定义，某款 ARM 芯片中，是否支持协处理器或支持哪个协处理器与 ARM 版本无关。生产商可以选择实现部分协处理器指令或者完全不支持协处理器。
- 如果协处理器必须完成一些内部工作来准备一个 32 位数据向 ARM 传送（例如，浮点 FIX 操作必须将浮点值转换为等效的定点值），那么这些工作必须在协处理器提交传送前进行。因此，在准备数据时经常需要协处理器握手信号处于“忙－等待”状态。ARM 可以

在忙—等待时间内产生中断。如果它确实得以中断，那么它将暂停握手以服务中断。当它从中断服务程序返回时，将可能重试协处理器指令，但也可能不重试。例如，中断可能导致任务切换。无论哪种情况，协处理器必须给出一致结果，因此，在握手提交阶段之前的准备工作不允许改变处理器的可见状态。

9.2 状态寄存器访问指令

ARM 指令集提供了两条指令，可直接控制程序状态寄存器 psr。MRS 指令用于把 cpsr 或者 spsr 的值传送到一个寄存器；MSR 与之相反，它将一个寄存器的内容传送到 cpsr 或者 spsr。这两条指令结合，可用于对 cpsr 和 spsr 进行读/写操作。

表 9.2 总结了状态寄存器访问指令。

表 9.2　　状态寄存器访问指令

助　记　符	含　　义	操　　作
MRS	将程序状态字寄存器的值送到通用寄存器	Rd=spr
MSR	将通用寄存器的值送到程序状态字寄存器	Psr[field]=Rm
MSR	将一个立即数送到程序状态字	Psr[field]=immediate

当需要保存或修改当前模式下 CISR 或 SPSR 的内容时，首先必须将这些内容传送到通用寄存器中，对选择的位进行修改，然后将数据回写到状态寄存器。对于 ARM 和 Thumb 的状态切换也是如此，程序不能通过直接改写 CPSR 中的 T 控制位直接将程序状态切换到 Thumb 状态，必须通过 BX 等指令完成程序状态的切换。

9.2.1 程序状态字内容送通用寄存器指令 MRS

1. 指令编码格式

读状态寄存器指令 MRS。在 ARM 寄存器中，只有 MRS 指令可以将状态寄存器 CPSR 或 SPSR 读出到通用寄存器中。将程序状态字内容读取到通用寄存器中后就可以对其进行计算、修改等操作。

指令的编码格式如图 9.6 所示。

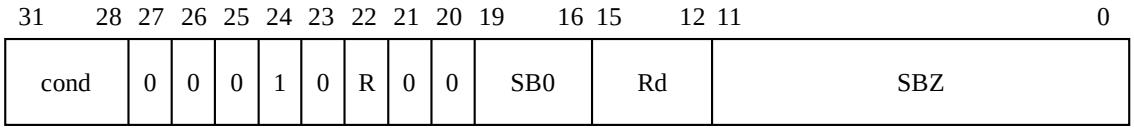


图 9.6　MRS 指令编码格式

2. 指令的语法格式

```
MRS{<cond>}  <Rd>，CPSR
```

```
MRS{<cond>}  <Rd>，SPSR
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定指令的目标寄存器。如果 r15 被用作目标寄存器，指令的执行结果不可预知。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    If  R==1  then
        Rd=SPSR
    Else
        Rd=CPSR
```

4．指令的使用

MRS 指令注意用于以下 3 种场合：

- 当需要保存或修改当前模式下 CPSR 或 SPSR 的内容时，首先必须将这些内容传送到通用寄存器中，对选择的位进行修改，然后将数据回写到状态寄存器。
- 当异常中断允许嵌套时，需要在进入异常中断之后，嵌套中断发生之前保存当前处理器模式对应的 SPSR。这时需要先通过 MRS 指令读出 SPSR 的值，再用其他指令（如压栈指令）将 SPSR 值保存起来。
- 在进程切换时也需要保存当前状态寄存器的值。

注意 在用户模式下对 CPSR[23：0]进行任何修改都是无效的。另外，尽量避免在用户模式或系统模式下访问 SPSR，因为在这种模式下没有 SPSR，如果执行此操作，指令的执行结果不可预知。

5．指令举例

（1）将 CPSR 状态寄存器读取，保存到 r1 中。

```
MRS   r1，CPSR
```

（2）将 SPSR 状态寄存器读取，保存到 r2 中。

```
MRS   r2，SPSR
```

（3）MSR 指令读取 CPSR，用来判断 ALU 的状态标志或 IRQ/FIQ 中断是否允许等；在异常处理程序中，读 SPSR 可知道进入异常前的处理器状态等。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读－修改－写操作，可用来进行处理器模式的切换或禁止/使能 IRQ/FIQ 中断等设置。另外，进行切换或允许异常中断嵌套时，也需要使用 MRS 指令读取

SPSR 状态值保存起来。

下面的例子使能 IRQ 中断。

```
ENABLE_IRQ
        MRS    r0，CPSR
        BIC    r0，r0，＃0x80
        MSR    CPSR_c，r0
        MOV    PC，LR
```

下面的例子禁止 IRQ 中断。

```
DISABLE_IRQ
       MRS    r0，CPSR
       ORR    r0，r0，＃0x80
       MSR    CPSR_c，r0
       MOV    PC，LR
```

9.2.2 写状态寄存器指令 MSR

1．指令编码格式

写状态寄存器指令 MSR（Move to Status Register from ARM Register）。在 ARM 处理器中，只有 MSR 指令可以直接设置状态寄存器 CPSR 或 SPSR。

图 9.7、图 9.8 分别显示了源操作数为立即数的 MSR 指令和源操作数为寄存器的 MSR 指令的编码格式。

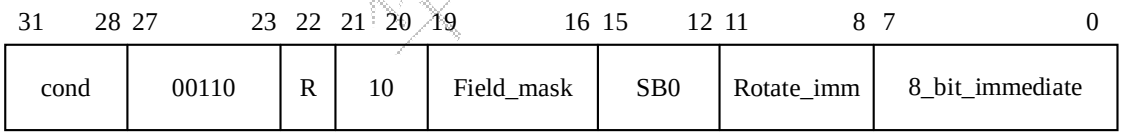


图 9.7　源操作数为立即数的 MSR 指令编码格式

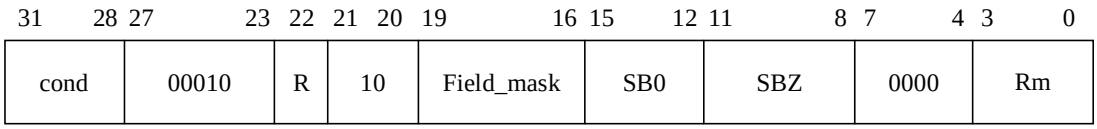


图 9.8　源操作数为寄存器的 MSR 指令编码格式

2．指令的语法格式

```
MSR{<cond>}   CPSR_<fields>，#<immediate>
MSR{<cond>}   CPSR_<fields>，<Rm>
MSR{<cond>}   SPSR_<fields>，#<immediate>
MSR{<cond>}   SPSR_<fields>，<Rm>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <fields>

域标志位，是下面选项中的一种或几种的组合。

- C：设置控制位掩码域（bit[16]）。
- X：设置扩展位掩码域（bit[17]）。
- S：设置状态位掩码域（bit[18]）。
- F：设置标志位掩码域（bit[19]）

③ <immediate>

将被传送到 CPSR 和 SPSR 寄存器的立即数。此立即数可以为 8 位立即数（范围在 0x00～0xff 之间）。

④ <Rm>

指定的通用寄存器，此寄存器包含将要被传送状态寄存器中的数据。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    If  opcode[25]==1
        Operand=8_bit_immediate  Rotate_Right{rotate_imm*2}
    Else /*opcode[25]==0*/
        Operand=Rm
    If  R==0 then
        If  field_mask[0]==1 and inAprivilegedMode()  then
            CPSR[7:0]=operand[7:0]
        If  field_mask[1]==1 and inAprivilegedMode()  then
            CPSR[15:8]=operand[15:8]
        If  field_mask[2]==1 and inAprivilegedMode()  then
            CPSR[23:16]=operand[23:16]
        If  field_mask[3]==1 and then
            CPSR[31:24]=operand[31:24]
    Else/*R==1*/
        If  field_mask[0]==1 and CurrentModeHasSPSR()  then
            SPSR[7:0]=operand[7:0]
        If  field_mask[1]==1 and CurrentModeHasSPSR()  then
            SPSR[15:8]=operand[15:8]
        If  field_mask[2]==1 and CurrentModeHasSPSR()  then
            SPSR[23:16]=operand[23:16]
        If  field_mask[3]==1 and CurrentModeHasSPSR()  then
            SPSR[31:24]=operand[31:24]
```

4．指令的使用

详见 MRS 指令。

5．指令举例

（1）使用“读－修改－写”模式更新 CPSR 寄存器。

```
MRS   r0，CPSR                  ;读 CPSR 寄存器的值
BIC   r0，r0，＃0xf0000000      ;清除 N、Z、C、V 位
MSR   CPSR_f，r0                ;更新 CPSR 中的标志位
```

注意 因为 PSR 状态寄存器中存在目前没有定义的保留位，所以在使用时，最好加上“_fsxc”后缀，如上例中的“CPSR_f”。这样做的目的是使指令只修改程序状态寄存器的某个域，防止程序向高版本指令集移植时发生意外。

（2）禁止 IRQ 中断。

```
MRS    r0，CPSR               ;读 CPSR 状态寄存器
ORR    r0，r0，＃0x80         ;设置 IRQ 中断禁止位
MSR    CPSR_c，r0             ;更新 CPSR 状态寄存器
```

（3）堆栈初始化。

```
INITSTACK
        MOV    r0，LR          ;保存返回地址
; 设置管理模式堆栈
        MSR    CPSR_c，＃0xd3;
        LDR    SP，StackSvc;
; 设置中断模式堆栈
        MSR    CPSR_c，＃0xd2;
        LDR    SP，StackIrq;
```

9.3 零计数指令 CLZ

ARMv5 及其以上版本提供了一条新的指令——零计数指令 CLZ（Count Leading Zeros）。该指令用于计算最高符号位与第一个 1 之间的 0 的个数。当一些操作数需要规范化（使其最高位为 1）时，该指令用于计算操作数需要左移的位数。

1．指令编码格式

CLZ 指令返回操作数二进制编码中第一个 1 前 0 的个数。如果操作数为 0，则指令返回 32；如果操作数二进制编码第 31 位为 1，指令返回 0。

指令编码格式如图 9.9 所示。

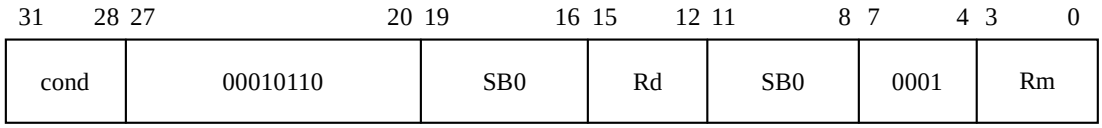


图 9.9 CLZ 指令的编码格式

注意 该指令不影响程序状态字的条件标志位。

2．指令的语法格式

```
CLZ{<cond>}   <Rd>，<Rm>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定指令的目标寄存器，如果 r15 用做目标寄存器，指令的执行结果不可预知。

③ <Rm>

确定指令的源寄存器，如果 r15 被用作源寄存器，指令的执行结果不可预知。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If   Rm==0
      Rd=32
Else
      Rd=31-(bit position of most significant “1” in Rm)
```

4．指令的使用

该指令一般用于下面两种情况。

- 计算操作数规范化时需要左移的位数。
- 确定一个优先级掩码中最高优先级（最高位的优先级）。

5．指令举例

下面的例子标准化 Rm 中的数据。注意其中的移位指令使用了 MOVS 而非 MOV，这主要是考虑到 Rm 中的数据为 0 的特殊情况。

```
CLZ   Rd，Rm
MOVS   Rm，Rm，LSL  Rd
```

9.4 交换指令

交换指令是 load/Store 指令的一种特殊形式。该指令将一个存储器单元内容与指定的寄存器内容相交换。交换指令为进程间同步提供了一种方便的解决途径。该指令产生一对原子 Load/Store 操作（an atomic load and store operation），该操作发生在一个连续的总线操作中，在操作期间阻止其他任何指令对该存储单元的读/写。

表 9.3 总结了 ARM 的交换指令。

表 9.3　　交换指令

助　记　符	含　　义	操　　作
SWP	寄存器和存储器字数据交换	Rd←[Rd]，[Rn]←[Rm](Rn≠Rd 或 Rm)
SWPB	寄存器和存储器字节数据交换	Rd←[Rd]，[Rn]←[Rm](Rn≠Rd 或 Rm)

9.4.1 寄存器和存储器字数据交换指令 SWP

1．指令编码格式

寄存器和存储器字交换指令 SWP（Swap）用于将一个内存单元（该单元地址放在寄存器 Rn 中）的内容读取到一个寄存器 Rd 中，同时将另一个寄存器 Rm 的内容写入到该内存单元中。

指令的编码格式如图 9.10 所示。

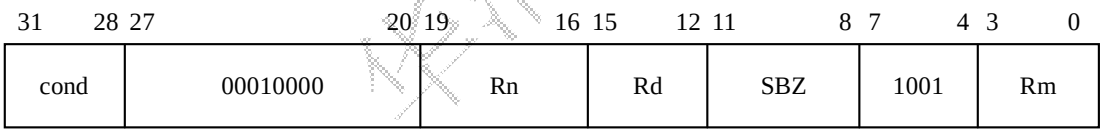


图 9.10　SWP 指令编码格式

2．指令的语法格式

```
SWP{<cond>}   <Rd>，<Rm>，[<Rn>]
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定指令的目标寄存器。

③ <Rm>

该寄存器包含将要被存储到内存单元中的数据。

④ <Rn>

内存单元地址寄存器。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    If  Rn[1:0]==0b00  then
        Temp=memory[Rn,4]
    Else if  Rn[1:0]==0b01  then
        Temp= memory[Rn,4]  Rotate_Right 8
    Else if  Rn[1:0]==0b10  then
        Temp= memory[Rn,4]  Rotate_Right 16
    Else   /*  Rn[1:0]==0b01  then  */
        Temp= memory[Rn,4]  Rotate_Right 24

    Memory[Rn,4]=Rm
    Rd=temp
```

4．指令举例

（1）将 r1 的内容与 r0 指向的存储单元的内容进行交换。

```
SWP    r1，r1，[r0]
```

（2）使用 SWP 指令进行信号量操作。

```
SEM       EQU       0x10002000
……
WAIT_SEM
          MOV    r0，＃0          ;
          LDR    r0，=SEM         ;
          SWP    r1，r1，[r0]     ;取出信号量
          CMP    r1，＃0          ;判断是否有信号
          BEQ    WAIN_SEM         ;若没有，继续等待
```

9.4.2　寄存器和存储器字节数据交换指令 SWPB

1．指令编码格式

寄存器和存储器字节交换指令 SWPB（Swap Byte）。将内存单元中一个字节的内容和寄存器内容进行交换。详情请参见 SWP 指令。

指令的编码格式如图 9.11 所示。

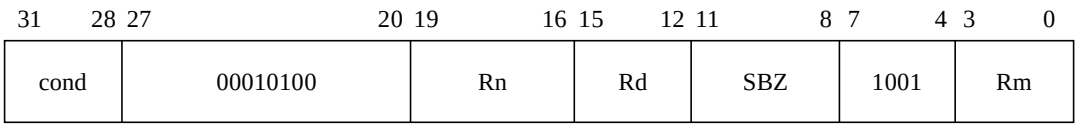


图 9.11　SWPB 指令编码格式

2．指令的语法格式

```
SWP{<cond>}B   <Rd>，<Rm>，[<Rn>]
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定指令的目标寄存器。

③ <Rm>

该寄存器包含将要被存储到内存单元中的数据。

④　<Rn>

内存单元地址寄存器。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Temp=Memory[Rn,1]
    Memory[Rn,1]=Rm[7:0]
    Rd=temp
```

4．指令举例

参见 SWP 指令。

9.5　异常产生指令

ARM 指令集中提供了两条产生异常的指令，通过这两条指令可以用软件的方法实现异常。表 9.4 总结了 ARM 异常产生指令。

表 9.4　　ARM 异常产生指令

助　记　符	含　　义	操　　作
SWI	软中断指令	产生软中断，处理器进入管理模式
BKPT	断点中断指令	处理器产生软件断点

9.5.1 软中断指令 SWI

1．指令编码格式

软件中断指令 SWI（Software Interrupt）用于产生软中断，从而实现从用户模式变换到管理模式，CPSR 保存到管理模式的 SPSR 中，执行转移到 SWI 向量，在其他模式下也可以使用 SWI 指令，处理器同样切换到管理模式。

指令的编码格式如图 9.12 所示。

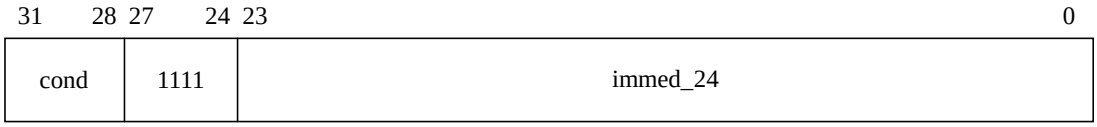


图 9.12 SWI 指令编码格式

2．指令的语法格式

```
SWI{<cond>}   <immed_24>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <immed_24>

指定一个 24 位立即数。ARM 处理器不对该立即数进行任何处理，其作用是提供给操作系统，从而判断用户程序请求的服务类型。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    R14_svc = address of next instruction after the SWI instruction
    SPSR_svc = CPSR
    CPSR[4:0] = 0b10011    /*进入超级用户模式*/
    CPSR[5] = 0    /*执行在 ARM 状态*/
    CPSR[7] = 1
    If  high  vectors configured then
        PC = 0xffff0008
    Else
        PC = 0x00000000
```

4．指令举例

（1）下面指令产生软中断，中断立即数为 0。

```
SWI   0;
```

（2）产生软中断，中断立即数为 0x123456。

```
SWI   0x123456;
```

（3）使用 SWI 指令时，通常使用以下两种方法进行参数传递。

① 指令 24 位的立即数指定了用户请求的类型，中断服务程序的参数通过寄存器传递。

下面的程序产生一个中断号为 12 的软中断。

```
MOV   r0，#34              ;设置功能号为 34
SWI   12                   ;产生软中断，中断号为 12
```

② 另一种情况，指令中的 24 位立即数被忽略，用户请求的服务类型由寄存器 r0 的值决定，参数通过其他寄存器传递。

下面的例子通过 r0 传递中断号，r1 传递中断的子功能号。

```
MOV   r0，#12              ;设置 12 号软中断
MOV   r1，#34              ;设置子功能号为 34
SWI    0                   ;
```

（4）在 SWI 异常中断处理程序中，取出 SWI 立即数的步骤为：首先确定引起软中断的 SWI 指令是 ARM 指令还是 Thumb 指令，这可通过对 SPSR 访问得到；然后要确定该 SWI 指令的地址，这可通过访问 LR 寄存器得到；然后读出指令，分解立即数。

下面的例子为一个标准的 SWI 中断处理程序。

```
T_bit     EQU     0x20
SWI_Hander
          STMFD   SP!,{r0_r3,r12,LR} ;保护现场
          MOV     r1，sp             ;设置参数指针
          MRS     r0，SPSR           ;读取 SPSR
          STMFD   SP!,{r0,r3}        ;保持 SPSR，r3 压栈保证字节对齐
          TST     r0,#T_bit          ;测试 T 标志位
          LDRNEH  r0,[LR,#-2]        ;若为 Thumb 指令，读取指令码（16 位）
          BICNE   r0,r0,#0xff00      ;取得 Thumb 指令 8 位立即数
          LDREQ   r0,[LR,#-4]        ;若为 ARM 指令，读取指令码（32 位）
          BICNQ   r0,r0,#0xff00000   ;取得 ARM 指令的 24 位立即数
        ; r0 存储中断号
        ; r1 指向栈顶

    BL     C_SWI_Handler             ;调用主要的中断服务程序
    LDMFD   sp!, {r0, r3}            ;SPSR 出栈
    MSR     spsr_cf, r0              ;恢复 SPSR
    LDMFD   sp!, {r0-r3, r12, pc}^   ;保存寄存器并返回
```

中断服务程序的主要工作放在 C_SWI_Handler 中，由 C 语言完成，用 swich_case 结构判断中断类型。典型的程序如下。

```
void C_SWI_Handler( int swi_num, int *regs )
{
     switch( swi_num )
     {
     case   0:
          regs[0] = regs[0] * regs[1];
     break;

     case   1:
          regs[0] = regs[0] + regs[1];
     break;

     case   2:
          regs[0] = (regs[0] * regs[1]) + (regs[2] * regs[3]);
     break;

     case   3:
     {
          int w, x, y, z;

          w = regs[0];
          x = regs[1];
          y = regs[2];
          z = regs[3];

          regs[0] = w + x + y + z;
          regs[1] = w - x - y - z;
          regs[2] = w * x * y * z;
          regs[3] =(w + x) * (y - z);
     }
     break;

     }
}
```

9.5.2 断点中断指令 BKPT

1．指令编码格式

断点中断指令 BKPT（BreakPoint）产生一个预取异常（prefetch abort），它常被用来设置软件断点，在调试程序时十分有用。当系统中存在调试硬件时，该指令被忽略。

指令的编码格式如图 9.13 所示。

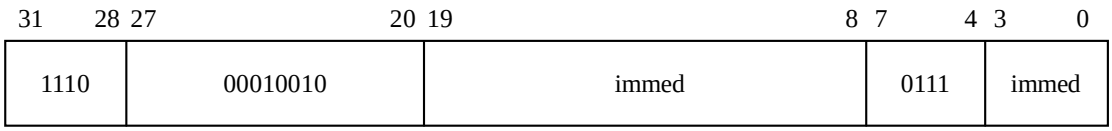


图 9.13 BKPT 指令编码格式

2．指令的语法格式

```
BKPT    <immediate>
```

<immediate>

16 位的立即数。该立即数可以用来保存额外的断点信息。

注意 16 位立即数在指令的编码格式中并不是连续存放的。前 12 位放在 bits[19：8]，而后 4 位放在 bits[3：0]。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  (not overridden by debug hard)
      R14_abt = address of BKPT instruction + 4
      SPSR_abt = CPSR
      CPSR[4:0] = 0b10111          /*进入异常模式*/
      CPSR[5] = 0                  /*执行在 ARM 状态*/
      /*CPSR[6] is unchanged*/
      CPSR[7] = 1                  /*禁止正常中断*/
      If  high vectors configured then
            PC = 0x0000000c
```

4．指令的使用

要正确的使用 BKPT 指令，必须和具体的调试系统相结合。一般说来，BKPT 有两种使用方法。

（1）如果当前使用的系统调试硬件没有屏蔽 BKPT 指令，那么在此系统中预取指令异常和软件调试命令同时使用一个中断向量。这样当异常发生时，就要依靠系统自身来判断是真正的预取异常还是软件调试命令。判断的方法，根据系统的不同，而有所不同。

（2）如果当前的系统调试硬件屏蔽了 BKPT 指令，那么系统会跳过 BKPT 指令顺序执行该指令下面的程序代码。

注意 BKPT 指令总是无条件执行的，当指令的编码格式中的条件域不被解析为 AL 时，指令的执行结果不可预知。

华清远见

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 10 章 ARM 汇编程序设计

华清远见

ARM 源程序文件（即源文件）有特定的文件格式和语法规则，可以使用任意文本编辑器编写程序代码。一般地，ARM 源程序文件名的后缀如表 10.1 所示。

表 10.1 ARM 源程序文件名后缀

程　　序	文　件　名
汇编	*.S
引入文件	*.INC
C 程序	*.C
头文件	*.H

在一个项目中，至少要有一个汇编源文件，可以有多个汇编源文件或多个 C 程序，或者 C 程序文件和汇编文件两者的组合。

ARM 汇编语言语句格式如下所示。

```
{label}{instruction/directive/pseudo-instruction}{;comment}[1]
```

注意 所有指令均不能顶格写，要用空格（space）或 TAB 开头。

其中 instruction 即 ARM 指令集中的汇编指令。Directive 为 ARM 汇编器所支持的伪操作。pseudo-instruction 为 ARM 汇编器所支持的伪操作。下面章节分别介绍伪操作和伪指令。

10.1 ARM 汇编器所支持的伪操作

在 ARM 汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪操作标识符（directive）[2]，它们所完成的操作称为伪操作。伪操作在源程序中的作用是为完成汇编程序作各种准备工作的，这些伪操作仅在汇编过程中起作用，一旦汇编结束，伪操作的使命就完成。

在 ARM 的汇编程序中，伪操作主要有符号定义伪操作、数据定义伪操作、汇编控制伪操作、宏指令等。

10.1.1 符号定义（Symbol Definition）伪操作

符号定义伪操作用于定义 ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。常见的符号定义伪操作有如下几种。

- 用于定义全局变量的 GBLA、GBLL 和 GBLS。
- 用于定义局部变量的 LCLA、LCLL 和 LCLS。

[1] 这里为保持和国内在 IBM PC 汇编语言中对名词翻译的一致性，derective 称为“伪操作”。同样在 ARM 中宏指令被称为 pseudo-insruction，这里将其称为“宏指令”，宏指令也是通过伪操作定义的。

[2] 有些文献中也称其为操作标识。

- 用于对变量赋值的 SETA、SETL、SETS。
- 为通用寄存器列表定义名称的 RLIST。
- 为协处理器寄存器定义别名的 CN。
- 为协处理器定义别名的 CP。
- 为 VFP 寄存器定义名称的 DN 和 SN。
- 为 FPA 浮点指针寄存器定义名称的 FPA。

1．全局变量定义伪操作 GBLA、GBLL 和 GBLS

（1）语法格式

GBLA、GBLL 和 GBLS 伪操作用于定义一个 ARM 程序中的全局变量并将其初始化。其中：

GBLA 伪操作用于定义一个全局的数字变量并初始化为 0。

GBLL 伪操作用于定义一个全局的逻辑变量并初始化为 F（假）。

GBLS 伪操作用于定义一个全局的字符串变量并初始化为空。

由于以上 3 条伪指令用于定义全局变量，因此在整个程序范围内变量名必须惟一。

语法格式如下。

```
<gblx>   <variable>
```

① <gblx>

取值为 GBLA、GBLL、GBLS 之一。

② <variable>

定义的全局变量名，在其作用范围内必须惟一。全局变量的作用范围为包含该变量的源程序。

（2）使用说明

如果用这些伪操作重新声明已经声明过的变量，变量的值将被初始化成后一次声明语句中的值。

（3）示例

① 使用伪操作声明全局变量。

```
    GBLA      Test1          ;定义一个全局的数字变量，变量名为 Test1
    Test1  SETA    0xaa      ;将该变量赋值为 0xaa
    GBLL      Test2          ;定义一个全局的逻辑变量，变量名为 Test2
    Test2  SETL    {TRUE}    ;将该变量赋值为真
    GBLS      Test3          ;定义一个全局的字符串变量，变量名为 Test3
    Test3  SETS    "Testing" ;将该变量赋值为“Testing”
```

② 声明变量 objectsize 并设置其值为 0xff，为“SPACE”操作做准备。

```
      GBLA        objectsize
Objectsize      SETA     oxff
```

```
        SPACE     objectsize
```

③ 下面的例子显示如何使用汇编命令设置变量的值。具体做法是使用“–pd”选项。

```
    Armasm –pd "objectsize SETA oxff" –o objectfile sourcefile
```

2．局部变量定义伪操作 LCLA、LCLL 和 LCLS

（1）语法格式

LCLA、LCLL 和 LCLS 伪指令用于定义一个 ARM 程序中的局部变量并将其初始化。其中：

LCLA 伪操作用于定义一个局部的数字变量并初始化为 0。

LCLL 伪操作用于定义一个局部的逻辑变量并初始化为 F（假）。

LCLS 伪操作用于定义一个局部的字符串变量并初始化为空。

以上三条伪操作用于声明局部变量，在其作用范围内变量名必须惟一。

语法格式如下。

```
<lclx>  <variable>
```

① <gblx>

取值为 LCLA、LCLL、LCLS 之一。

② <variable>

所定义的局部变量名，在其作用范围内必须惟一。局部变量作用范围为包含该局部变量的宏。

（2）使用说明

如果用这些伪操作重新声明已经声明过的变量，则变量的值将被初始化成后一次声明语句中的值。

（3）示例

① 使用伪操作声明局部变量。

```
    LCLA      Test4                ;声明一个局部的数字变量，变量名为 Test4
    Test3  SETA    0xaa            ;将该变量赋值为 0xaa
    LCLL      Test5                ;声明一个局部的逻辑变量，变量名为 Test5
    Test4  SETL    {TRUE}          ;将该变量赋值为真
    LCLS      Test6                ;定义一个局部的字符串变量，变量名为 Test6
    Test6  SETS   "Testing"        ;将该变量赋值为 “Testing”
```

② 下面的例子定义一个宏，显示了局部变量的作用范围。

```
      MACRO                        ;声明一个宏
$label   message  $a               ;宏原型
     LCLS   err                    ;声明局部字符串变量
$label
     INFO       0,"err":CC::STR:$a
```

```
    MEND                                ;宏结束，局部变量不再起作用
```

3．变量赋值伪操作 SETA、SETL 和 SETS

（1）语法格式

伪指令 SETA、SETL 和 SETS 用于给一个已经定义的全局变量或局部变量赋值。

SETA 伪操作用于给一个数学变量赋值；

SETL 伪操作用于给一个逻辑变量赋值；

SETS 伪操作用于给一个字符串变量赋值；

语法格式如下。

```
Variable  <setx>  expr
```

① Variable

变量名为已经定义过的全局变量或局部变量，表达式为将要赋给变量的值。

② <setx>

取值为 SETA、SETL、SETS 之一。

③ expr

数学、逻辑或字符串表达式，也就是将要赋予变量的值。

（2）使用说明

在向变量赋值前必须先声明变量。

也可以在汇编指令中预定义变量，如：

```
"Armasm  --pd  "objectsize  SETA  oxff"  --o  objectfile  sourcefile"
```

（3）示例

① 为预先定义的变量赋值。

```
    LCLA      Test3             ;声明一个局部的数字变量，变量名为 Test3
    Test3  SETA  0xaa           ;将该变量赋值为 0xaa
    LCLL      Test4             ;声明一个局部的逻辑变量，变量名为 Test4
    Test4  SETL  {TRUE}         ;将该变量赋值为真
    LCLS      Test6             ;定义一个局部的字符串变量，变量名为 Test6
    Test6  SETS   "Testing"     ;将该变量赋值为 “Testing”
```

② 使用变量赋值伪操作，定义一些程序相关内容。

```
    GBLA            versionNumber
    VersionNumber     SETA    21

    GBLL            Debug
    Debug      SETL  {TRUE}

    GBLS          versionString
```

```
    VersionString     SETS   "version 1.0"
```

4．通用寄存器列表定义伪操作 RLIST

（1）语法格式

RLIST 伪操作可用于对一个通用寄存器列表定义名称，使用该伪操作定义的名称可在 ARM 指令 LDM/STM 中使用。在 LDM/STM 指令中，列表中的寄存器访问次序根据寄存器的编号由低到高，与列表中的寄存器排列次序无关。

语法格式如下。

```
Name  RLIST  {list-of-registers}
```

① Name

寄存器列表的名称。

注意 该名称不能和已经定义寄存器或协处理器名称相同。

② list-of-registers

通用寄存器列表。列表中的寄存器用“,”隔开，如果是编号连续的通用寄存器可以用“–”指定寄存器范围。具体用法参见程序示例。

（2）使用说明

在使用 ARM 汇编编译器编译源文件时，可以使用“–checkreg”选项来指定汇编器进行寄存器检查。如果汇编器检测到寄存器列表中的寄存器编号非升序排列，将给出编译警告。

（3）示例

① 将寄存器列表名称定义为 RegList，可在 ARM 指令 LDM/STM 中通过该名称访问寄存器列表。

```
RegList  RLIST  {R0-R5，R8，R10}    ;
```

② 使用“–”在寄存器列表中，指定寄存器范围。

```
Context  RLIST  {r0-r6,r8,r10-r12,r15}  ;
```

5．协处理器寄存器名称定义伪操作 CN

（1）语法格式

CN 伪操作为协处理器寄存器定义名称。

语法格式如下。

```
Name  CN  expr
```

① Name

定义的协处理器寄存器的名称。

注意 该名称不能和已经定义寄存器或协处理器名称相同。

② expr
协处理器寄存器编号。
（2）使用说明
协处理器寄存器编号的数值范围为 0～15。避免使用不同的名称定义同一物理寄存器。

注意 协处理器寄存器的名称不能被定义为 c0～c15，这些名称已经被汇编器预定义。

（3）示例
将协处理器寄存器 6 命名为 Power。

```
Power    CN  6
```

6．协处理器名称定义伪操作 CP

（1）语法格式
CP 伪操作为指定的协处理器定义名称。
语法格式如下。

```
Name   CP  expr
```

① Name
定义的协处理器名称。

注意 该名称不能和已经定义寄存器或其他协处理器名称相同。

② expr
协处理器编号。
（2）使用说明
协处理器编号范围为 0～15。
使用 CP 伪操作为协处理器定义一个方便记忆的名称，可以使程序员更高效地编写代码。

注意 协处理器寄存器的称不能被定义为 p0～p15，这些名称已经被汇编器预定义。

（3）示例
将协处理器 6 命名为 Dmu。

```
Dmu   CP  6
```

7．VFP 寄存器名称定义伪操作 DN/SN

（1）语法格式

DN 伪操作为双精度（double-precision）VFP 寄存器定义名称。D0～D15 是汇编器预先定义的，用户不能使用。

SN 伪操作为单精度（single-precision）VFP 寄存器定义名称。S0～S31 是汇编器预先定义的，用户不能使用。

语法格式如下。

```
Name  DN  expr
Name  SN  expr
```

① Name

指定的 VFP 寄存器的名称。



注意 该名称不能和已经定义寄存器或其他协处理器名称相同。

② expr

指定 VFP 寄存器编号。对于双精度寄存器编号范围为 0～15；对于单精度寄存器编号范围为 0～31。

（2）示例

① 将 VFP 双精度寄存器 6 定义为 energy。

```
energy    DN  6
```

② 将 VFP 单精度寄存器 16 定义为 mass。

```
mass  SN  16
```

8．浮点寄存器名称定义伪操作 FN

（1）语法格式

FN 为一个 FPA 浮点寄存器定义名称。F0～F7 是汇编器预先定义的，用户不能使用。



注意 FPA 的使用在 ARM 公司新发布的编译器 RVCT 中已不再支持。

语法格式如下。

```
Name  FN  expr
```

① Name

指定的浮点寄存器的名称。



注意 该名称不能和已经定义寄存器或其他协处理器名称相同。

② expr

指定浮点寄存器编号。编号范围为 0～7。

（2）示例

为浮点寄存器 6 指定名称为 Energy。

```
Energy   FN  6
```

10.1.2 数据定义（Data Definition）伪操作

数据定义伪操作一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。常见的数据定义伪操作有如下几种。

- DCB 用于分配一片连续的字节存储单元并用指定的数据初始化。
- DCW（DCWU）用于分配一片连续的半字存储单元并用指定的数据初始化。
- DCD（DCDU）用于分配一片连续的字存储单元并用指定的数据初始化。
- DCFD（DCFDU）用于为双精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
- DCFS（DCFSU）用于为单精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
- DCQ（DCQU）用于分配一片以 8 字节为单位的连续的存储单元并用指定的数据初始化。
- SPACE 用于分配一片连续的存储单元。
- MAP 用于定义一个结构化的内存表首地址。
- FIELD 用于定义一个结构化的内存表的数据域。
- LTORG 用于声明一个数据缓冲池（literal pool）的开始。
- DCDO 用于分配一段字的内存单元，并将单元内容初始化为该单元相对于静态基址寄存器（r9）的偏移量。
- DCI 在 ARM 代码中，该伪操作分配一定字的内存单元；在 Thumb 代码中，该伪操作分配一定的半字内存单元。由伪操作 DCI 定义的内存单元存放的是代码而不是数据。
- COMMON 用于定义一块连续的内存，该内存单元的大小由用户指定。
- DATA 在代码中使用数据。现已不再使用，仅用于保持向前兼容。如果在源文件中出现，将被汇编器忽略。

1．用于分配字节存储单元的伪操作 DCB

（1）语法格式

DCB 伪操作用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为数字或字符串。DCB 也可用“=”代替。

语法格式如下。

```
{label}  DCB  expr{,expr}
```

① {label}

程序标号。

② expr

可以是−128～255 的数字，也可以是字符串。

（2）使用说明

在使用 DCB 伪操作时，其后常跟 ALIGN 伪操作以保证内存地址对齐。

（3）示例

① 分配一片连续的字节存储单元并初始化为指定字符串。

```
Str  DCB  "This is a test! "    ;
```

② 与 C 中的字符串不同，ARM 汇编中的字符串不以 *null* 结尾，下面指令以 ARM 汇编形成一个 C 语言风格的字符串。

```
C_string  DCB  "C_string",0
```

2．用于分配半字存储单元的伪操作 DCW（DCWU）

（1）语法格式

DCW（或 DCWU）伪操作用于分配一片连续的半字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。

用 DCW 分配的字存储单元是半字对齐的，而用 DCWU 分配的字存储单元并不严格半字对齐。

语法格式如下。

```
{label}  DCW  expr{,expr}…
```

① {label}

程序标号，可选。

② expr

数字表达式，取值范围为−32768～65525。

（2）使用说明

DCW 可能在分配的内存单元前加一个字节以保证内存半字对齐。当程序对内存对齐方式要求不严格时可以是 DCWU 伪操作。

（3）示例

① 分配一片连续的半字存储单元并初始化。

```
DataTest  DCW    1，2，3;
```

② 在指定内存单元初始值时可以使用已定义的变量。

```
Data     DCW-255，2*number;
         DCWU    number+4;
```

3．用于分配字存储单元的伪操作 DCD（DCWU）

（1）语法格式

DCD（或 DCDU）伪操作用于分配一片连续的字存储单元并用伪指令中指定的表达式初

始化。其中，表达式可以为程序标号或数字表达式。DCD 也可用“&”代替。

用 DCDU 分配的字存储单元是字对齐的，而用 DCDU 分配的字存储单元并不严格字对齐。

语法格式如下。

```
{label}  DCD{U}  expr{,expr}
```

① {label}

程序标号，可选。

② expr

expr 可以是数字表达式或程序相关表达式（program-relative expression）

（2）使用说明

DCD 可能在分配的内存单元前加 1~3 字节以保证内存字对齐。当程序对内存对齐方式要求不严格时可以是 DCDU 伪操作。

（3）示例

① 分配一片连续的字存储单元并初始化。

```
DataTest  DCD  4，5，6;
```

② 用程序标号初始化内存单元。

```
DataTest  DCD  mem06+4
```

③ 在内存单元不能字对齐的情况下，使用 DCDU 伪操作。

```
        AREA  Mydata,  DATA,READWRITE
        DCB  255            ;字节定义使内存单元不能字对齐
Data3   DCDU  1,5,20;
```

4．用于为单精度浮点数分配内存单元的伪操作 DCFS（或 DCFSU）

（1）语法格式

DCFS（或 DCFSU）伪指令用于为单精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个单精度的浮点数占据一个字单元。

用 DCFS 分配的字存储单元是字对齐的，而用 DCFSU 分配的字存储单元并不严格字对齐。

语法格式如下。

```
{label}  DCFS{U}  fpliteral{,fpliteral}
```

① {label}

程序标号，可选。

② fpliteral

单精度浮点数

（2）使用说明

DCFS 可能在分配的内存单元前加 1～3 字节以保证内存字对齐。当程序对内存对齐方式要求不严格时可以是 DCFSU 伪操作。

此伪操作使用的单精度浮点数的范围为：1.17549435e −38～3.40282347e+38。

（3）示例

① 分配一片连续的字存储单元并初始化为指定的单精度浮点数。

```
FDataTest  DCFS  2E5，−5E−7   ;
```

② 分配一片连续的字存储单元并初始化为单精度浮点数，但不严格要求字对齐。

```
DCFSU  1.0,−0.1,3.1e6;
```

5．用于为双精度浮点数分配内存单元的伪操作 DCFD（或 DCFDU）

（1）语法格式

DCFD（或 DCFDU）伪指令用于为双精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个双精度的浮点数占据两个字单元。

用 DCFD 分配的字存储单元是字对齐的，而用 DCFDU 分配的字存储单元并不严格字对齐。

语法格式如下。

```
{label}  DCFD{U}  fpliteral{,fpliteral}
```

① {label}

程序标号，可选。

② fpliteral

双精度浮点数

（2）使用说明

DCFS 可能在分配的内存单元前加 1～3 字节以保证内存字对齐。当程序对内存对齐方式要求不严格时可以是 DCFSU 伪操作。

当程序中的浮点数要由 ARM 处理器进行操作时，用户选择的浮点处理器结构会自动完成字节顺序的转换。当编译时使用了编译选项–fpunone，伪操作 DCFS（DCFSU）不可使用。

此伪操作使用的单精度浮点数的范围为： 2.22507385850720138e−308 ～1.79769313486231571e+308。

（3）示例

① 分配一片连续的字存储单元并初始化为指定的双精度浮点数。

```
FDataTest    DCFD   2E115，-5E7    ;
```

② 分配一片连续的字存储单元并初始化为双精度浮点数，但不严格要求字对齐。

```
DCFDU   1.0,-0.1,3.1e6;
```

6．分配以 8 个字节为单位的连续存储区域的伪操作 DCQ(或 DCQU)

（1）语法格式

DCQ（或 DCQU）伪指令用于分配一片以 8 个字节为单位的连续存储区域并用伪指令中指定的表达式初始化。

用 DCQ 分配的存储单元是字对齐的，而用 DCQU 分配的存储单元并不严格字对齐。

语法格式如下。

```
{label}  DCQ{U}  {-}literal{,{-}literal}
```

① {label}

程序标号，可选。

② literal

用于初始化内存的数字必须是可数的数字表达式，其取值范围为 0～2^{64}–1。

可以在数字表达式前加负号来表示用负数初始化内存单元，但此时数字表达式的取值范围为–2^{63}～1。

（2）使用说明

DCQ 可能在分配的内存单元前加 1~3 字节以保证内存字对齐。当程序对内存对齐方式要求不严格时可以是 DCQU 伪操作。

（3）示例

① 分配一片连续的存储单元并初始化为指定的值。

```
DataTest  DCQ    100    ;
```

② 使用标号定义内存单元。

```
ECQU   number+4
```

7．内存单元分配伪操作 SPACE

（1）语法格式

SPACE 伪指令用于分配一片连续的存储区域并初始化为 0。其中，表达式为要分配的字节数。SPACE 也可用“%”代替。

语法格式如下。

```
{label}  SPACE  expr
```

① {label}

程序标号，可选。

② expr

分配的字节数。

（2）使用说明

SPACE 伪操作常和 ALIGN 一起使用，详见 ALIGN 伪操作。

（3）示例

① 分配连续 100 字节的存储单元并初始化为 0。

```
DataSpace SPACE    100  ;
```

② 在 Mydata 段的开始可以 255 个初始化为 0 的字节单元。

```
      AREA    Mydata,DATA,READWRITE
data1 SPACE   255;
```

8．定义结构化内存表首地址伪操作 MAP

（1）语法格式

MAP 伪操作用于定义一个结构化的内存表的首地址。MAP 也可用“ ^ ”代替。

表达式可以为程序中的标号或数学表达式，基址寄存器为可选项，当基址寄存器选项不存在时，表达式的值即为内存表的首地址，当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和。

MAP 伪操作通常与 FIELD 伪操作配合使用来定义结构化的内存表。

语法格式如下。

```
MAP  expr{,base-register}
```

① expr

如果基地址寄存器（base-register）没有指定，expr 表达式存储到结构化内存表首地址。如果表达式 expr 是“程序相关的（program-relative）”，则程序标号在使用前必须定义。

② base-register

指定一个寄存器。当指令中包含这一项时，结构化内存表的首地址为 expr 和 base-register 寄存器值的和。

（2）使用说明

MAP 伪指令通常与 FIELD 伪指令配合使用来定义结构化的内存表。

当基地址寄存器（base-register）一旦被指定，下面所有的 FIELD 伪操作全部以基地址为基础增加偏移量。

（3）示例

① 定义结构化内存表首地址的值为 0x100+R0。

```
MAP   0x100，R0    ;
```

② 不存在基地址寄存器，结构化内存表的首地址直接由表达式定义。

```
MAP    0;
```

9．定义结构化内存表中数据域的伪操作 FILED

（1）语法格式

FIELD 伪操作用于定义一个结构化内存表中的数据域。FILED 也可用“#”代替。

表达式的值为当前数据域在内存表中所占的字节数。

FIELD 伪操作常与 MAP 伪操作配合使用来定义结构化的内存表。MAP 伪操作定义内存表的首地址，FIELD 伪操作定义内存表中的各个数据域，并可以为每个数据域指定一个标号供其他的操作引用。

注意 MAP 和 FIELD 伪操作仅用于定义数据结构，并不实际分配存储单元。

语法格式如下。

```
{label}  FIELD  expr
```

① {label}

程序标号，可选。当指令中存在这一项时，label 的值为当前内存表的位置计数器{VAR}的值。汇编器处理完这条 FIELD 指令后，内存表计数器的值将加上 expr 的值。

② expr

FIELD 指定的域所占内存单元字节数。

（2）使用说明

MAP 伪操作中的基地址寄存器（base-register）一旦指定，将被其后的所有 FIELD 伪操作定义的数据域默认使用，指定遇到下一个包含基地址寄存器（base-register）的 MAP 指令。另外在操作中定义的标号可以被 LOAD/STORE 指令直接引用。

（3）示例

① 下面的例子定义了一个内存表，其首地址为固定地址 0x100，该结构化内存表包含 3 个域，A 的长度为 16 个字节，位置为 0x100，B 的长度为 32 个字节，位置为 0x110，S 的长度为 256 个字节，位置为 0x130。

```
MAP    0x100                    ;定义结构化内存表首地址的值为 0x100。
       A      FIELD    16       ;定义 A 的长度为 16 字节，位置为 0x100
       B      FIELD    32       ;定义 B 的长度为 32 字节，位置为 0x110
       S      FIELD    256      ;定义 S 的长度为 256 字节，位置为 0x130
```

② 下面的例子显示了一个寄存器相关的首地址定义结构化内存表。

```
    MAP 0,r9              ;将结构化内存表的首地址设为 r9 的值
    FIELD   4             ;
LAB FIELD   4             ;
    LDR r0,LAB            ;
```

最后的 LDR 指令，相当于：

```
LDR r0,[r9,#4]
```

10．声明数据缓存池的伪操作 LTORG

（1）语法格式

LTORG 伪操作用于声明一个数据缓冲池，在使用 LDR 伪指令时，要在适当的地方加入 LTORG 声明数据缓冲池，这样就会把要加载的数据保存到缓存池中，再使用 ARM 的加载指令读出数据（如果没有使用 LTORG 声明缓存池，则汇编器会在程序末尾自动声明）。

语法格式如下。

```
LTORG
```

（2）使用说明

ARM 汇编器会将缓存池自动放在代码段之后。当程序中使用 LDR、LDFD 指令时可能会发生数据越界，为了防止越界，可以使用 LTORG 伪操作定义数据缓冲池。

通常大的代码段可以使用多个缓存池以防止越界。

LTORG 伪操作一般放在无条件跳转指令之后或子程序返回指令之后，这样处理器就不会将缓存池中的内容当指令执行。

汇编器完成内存字对齐时也要用到缓冲池。

（3）示例

① 声明一个数据缓存池用来存储 0x12345678。

```
LDR r0,=0x12345678;
ADD r1,r1,r0;
MOV   PC,LR;
LTORG
……
```

② 在代码段中使用数据缓冲池。

```
    AREA  Example,CODE,READONLY
Start  BL  func1;
func1                       ;程序主体
;程序代码
    LDR r1,=0x55555555
    MOV PC,LR               ;子程序结束
    LTORG
Data    SPACE    4200       ;清除 4200 个内存字节
END
```

11．将内存单元的内容初始化为相对地址的伪操作 DCDO

（1）语法格式

DCDO 用于分配一段字内存单元，并将每个单元的内容初始化为该单元相对于静态基地址寄存器的偏移量。DCDO 伪指令作为基于静态基地址寄存器 r9 的偏移量分配内存单元。DCDO 伪指令分配的内存要求字对齐。

语法格式如下。

```
{label}  DCDO  expr{,expr}…
```

① {label}标号

程序标号，可选。

② expr

寄存器相关表达式或已定义的地址标号。

（2）使用说明

DCDO 伪操作为基于静态基址寄存器 r9 的偏移量分配内存单元。

（3）示例

分配 32 位的字单元，其值为 externsym 基于 r9 的偏移量。

```
DCDO  externsym;
```

12．分配用于存放代码的内存单元伪指令 DCI

（1）语法格式

在 ARM 代码中，DCI 用于分配一段字的内存单元，用指定的数据初始化。指定内存单元存放的是代码，而不是数据。

在 Thumb 代码中，DCI 用于分配一段半字节的内存单元，用指定的数据初始化。指定的内存单元存放的是代码，而不是数据。

语法格式如下。

```
{label}  DCI  expr
```

① {label}标号

内存块起始地址标号。

② expr

指定可数的数字表达式。

（2）使用说明

DCI 伪操作和 DCD 伪操作非常相似，不同之处在于 DCI 分配的内存中的数据被表示为指令，可用于通过宏操作来定义处理器不支持的指令。

DCI 伪操作要求内存对齐，对于 ARM 指令要求 4 字节对齐，对于 Thumb 指令要求 2 字节对齐。

（3）示例

下面的程序通过宏操作来定义处理器不支持的指令。

```
MACRO
Newinst  $Rd,$Rm    ;
DCI    0xe16f0f10:OR($Rd:SHL:12):OR:$Rm
MEND
```

13．用于分配由用户指定大小的内存单元的伪操作 COMMON

（1）语法格式

COMMON 用于定义一块连续的内存地址单元。用户可以根据需要定义此内存单元采用的对齐方式。默认的对齐方式是字对齐的。

语法格式如下。

```
COMMON  symbol{, size{, alignment}}
```

① symbol

用于定义内存单元的符号名，该名称是大小写敏感的。

② size

用于指定所有保留的字节数。如果不指定此变量，汇编器默认为 0。

③ alignment

指定内存对齐方式。

（2）使用说明

连接器将 COMMON 指定的内存单元作为 ZI 段初始化。

（3）示例

下面的例子定义大小为 255 字节的内存单元，该内存单元是字对齐的。

```
COMMON    xyz,255,4    ;
```

10.1.3 汇编控制（Assembly Control）伪操作

汇编控制伪操作用于控制汇编程序的执行流程，常用的汇编控制伪操作包括以下几条：

- IF、ELSE、ENDIF。
- WHILE、WEND。
- MACRO、MEND。
- MEXIT。

1. IF、ELSE、ENDIF

（1）语法格式

IF、ELSE、ENDIF 伪操作能根据条件的成立与否决定是否执行某个指令序列。当 IF 后面的逻辑表达式为真，则执行 IF 后的指令序列，否则执行 ELSE 后的指令序列。其中，ELSE 及其后指令序列可以没有，此时，当 IF 后面的逻辑表达式为真，则执行指令序列，否则继续执行后面的指令。

IF、ELSE、ENDIF 伪指令可以嵌套使用。

语法格式如下。

```
IF  logical-expressing
…….
{ELSE
……}
ENDIF
```

logical-expression

用于决定指令执行流程的逻辑表达式。

（2）使用说明

当程序中有一段指令需要在满足一定条件时执行，使用该指令。

该操作还有另一种形式。

```
IF  logical-expression
    Instruction
ELIF  logical-expression2
    Instructions
ELIF  logical-expression3
    Instructions
ENDIF
```

ELIF 形式避免了 IF-ELSE 形式的嵌套，使程序结构更加清晰、易读。

（3）示例

```
IF  {CONFIG}=16
    BNE_rt_udiv_1    ;
    LDR r0,=_rt_div0   ;
    BX  r0           ;
    ELSE
    BEQ_rt_div()      ;
ENDIF
```

2．WHILE、WEND

（1）语法格式

WHILE、WEND 伪操作能根据条件的成立与否决定是否循环执行某个指令序列。当 WHILE 后面的逻辑表达式为真，则执行指令序列，该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，直到逻辑表达式的值为假。

WHILE、WEND 伪指令可以嵌套使用。

语法格式如下。

```
WHILE  logical-expression
code
WEND
```

logical-expression

用于决定指令执行流程的逻辑表达式。

（2）使用说明

WHILE、WEND 指令形式在进入循环之前判断执行条件，如果在第一次进入循环时，逻辑表达式即为“假”，循环体可以不执行。

（3）示例

下面的例子用 count 来控制循环体执行次数。

```
Count    SETA    1        ;
```

```
        WHILE   count<5     ;
Count   SETA    count+1     ;
…
…
WEND
```

3．MACRO、MEND

（1）语法格式

MACRO、MEND 伪操作可以将一段代码定义为一个整体，称为宏指令，然后就可以在程序中通过宏指令多次调用该段代码。其中，$标号在宏指令被展开时，标号会被替换为用户定义的符号。

宏操作可以使用一个或多个参数，当宏操作被展开时，这些参数被相应的值替换。

宏操作的使用方式和功能与子程序有些相似，子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场，从而增加了系统的开销，因此，在代码较短且需要传递的参数较多时，可以使用宏操作代替子程序。

包含在 MACRO 和 MEND 之间的指令序列称为宏定义体，在宏定义体的第一行应声明宏的原型（包含宏名、所需的参数），然后就可以在汇编程序中通过宏名来调用该指令序列。在源程序被编译时，汇编器将宏调用展开，用宏定义中的指令序列代替程序中的宏调用，并将实际参数的值传递给宏定义中的形式参数。

MACRO、MEND 伪操作可以嵌套使用。

语法格式如下。

```
 MACRO
{$label} macroname {$parameter{,$parameter}…}
 ;code
 MEND
```

① {$label}

$标号在宏指令被展开时，标号会被替换为用户定义的符号。通常，在一个符号前使用“$”表示该符号被汇编器编译时，使用相应的值代替该符号。

② macroname

所定义的宏的名称。

③ $parameter

宏指令的参数。当宏指令被展开时将被替换成相应的值，类似于函数中的参数。

（2）使用说明

在子程序代码比较短而需要传递的参数比较多的情况下可以使用宏汇编技术。首先要先用 MACRO 和 MEND 伪操作定义宏，包括宏定义体代码。在 MACRO 伪操作之后的第一行声明宏的原型，其中包含该宏定义的名称及需要的参数。在汇编中可以通过该宏定义的名称来调用它。当源程序被编译时，汇编器将展开每个宏调用，用宏定义体代替源程序中宏定义的名称，并用实际参数值代替宏定义时的形式参数。

（3）示例

① 没有参数的宏定义如下。

```
MACRO
CSI_SETB                ;宏名为 CSI_SETB，无参数
LDR r0,=rPDATG          ;读取 GPGO 口的值
LDR r1,[r0]
LDR r1,r1,#0x01         ;CSI 置位
STR   r1,[r0]           ;输出控制
MEND
```

② 带参数的宏定义如下。

```
MACRO
$IRQ_Label       HANDLER    $IRQ_Exception
                 EXPORT     $IRQ_Label
                 IMPORT     $IRQ_Exception
$IRQ_Label
         SUB      LR,LR,#4
         SEMFD    SP!,{r0-r3,r12,LR}
         MRS r3,STSR
         STMFD    SP!.{r3}
……
MEND
```

③ 下面的程序显示了一个完整的宏定义和调用过程。

```
;宏定义
        MACRO                   ;开始宏定义
$label  mymacro  $p1,$p2
;code
$label.loop1                    ;代码段
; code
BGE $label.loop1
$label.loop2                    ;代码段
BL       $p1
BGT $label.loop2
;代码段
ADR $p2
;代码段
MEND
```

```
;程序汇编后，宏展开结果
abc       mymacro    subr1,de    ;使用宏
          ;代码段
abcloop1  ;代码段
          ;代码段
          BGE abcloop1
Abcloop2  ;代码段
          BL    subr1
          BGT abcloop2
          ;代码段
          ADR de
          ;代码段
```

4．MEXIT

（1）语法格式

MEXIT 用于从宏定义中跳转出去。

语法格式如下。

```
MEXIT
```

（2）示例

```
    MACRO
$abc    macro    abc    $paraml,$param2
;code
        WHILE    condition1
        ;code
        IF    condition2
              ;代码段
              MEXIT
        ELSE
              ;代码段
        ENDIF
        WEND
        ;代段段
        MEND
```

5．关于伪操作的嵌套

下面的伪操作在使用时可以嵌套，嵌套的深度不能超过 256。

- MACRO 宏定义。

- WHILE…WEND 循环。
- IF…ELSE…ENDIF 条件语句。
- INCLUDE 指定头文件。

当这些伪操作混合使用时，总的嵌套深度不能超过 256。

10.1.4 信息报告（Reporting）伪操作

信息报告伪操作用于汇编报告指示。该类伪操作如下。

- ASSERT 用于断言错误。
- INFO 用于汇编诊断信息显示。
- OPT 用于设置列表选项。
- TTL 和 SUBT 用于插入标题。

1．断言错误伪操作 ASSERT

（1）语法格式

ASSERT 为断言错误伪操作。在汇编器对汇编程序进行第二遍扫描时，如果发现 ASSERT 条件不成立，汇编器将报告错误信息。

语法格式如下。

```
ASSERT  logical-expression
```

其中 logical-expression 用于断言的逻辑表达式，其值为“真”或“假”。

（2）使用说明

ASSERT 伪操作用于确保源程序在汇编时满足一定的条件。当条件不成立，即逻辑表示 logical-expression 为“假”时，汇编器报告错误。

（3）示例

① 下面的程序在 Top 和 Temp 相等时报告错误。

```
ASSERT       Top<>Temp    ;
```

② 当 label1 代表的地址大于 label2 所代表的地址时报告错误。

```
ASSERT       label1<=label2    ;
```

2．诊断信息显示伪操作 INFO 或“!”

（1）语法格式

汇编诊断信息显示伪操作 INFO 用于在汇编器处理过程中的第一遍扫描或第二遍扫描时报告诊断信息。

“!”和 INFO 相似。

语法格式如下。

```
INFO  numeric-expression，string-expression
```

① numeric-expression

数字表达式，在汇编时计算。如果 numeric-expression 的值为 0，则通过第一遍汇编并在第二遍汇编时报告“string-expression”的内容；如果 numeric-expression 的值不等于 0，则在第一遍汇编过程中报告“string-expression”的内容并中止汇编。

② string-expression

字符串表达式，用于在汇编过程中报告信息。

（2）使用说明

INFO 提供了一种方便的方法创建用户自己的诊断信息。

（3）示例

下面的程序在第二遍汇编扫描时报告版本信息，并判断 cont1 和 cont2 的关系。

```
INFO    0,"verion  1.0"          ;在第二遍扫描时，报告版本信息
IF  cont1>cont2                  ;如果 cont1>cont2
    INFO    1,"cont1>cont2"      ;则在第一遍扫描时报告 "cont1>cont2"
ENDIF
```

3．设置列表选项伪操作 OPT

（1）语法格式

可以通过设置列表选项伪操作 OPT 在源程序中设置列表选项。

语法格式如下。

```
OPT  n
```

其中 *n* 所设置的选项编码如表 10.2 所示。

表 10.2　　OPT 伪操作选项编码

选项编码 *n*	选 项 含 义
1	打开常规列表选项
2	关闭常规列表选项
4	设置分页符，在新的一页开始显示
8	将行号重置为 0

续表

选项编码 *n*	选 项 含 义
16	打开 SET、GBL、LCL 伪操作显示开关
32	关闭 SET、GBL、LCL 伪操作显示开关
64	打开宏展开（macro expansions）显示开关
128	关闭宏展开显示开关
256	打开宏调用（macro invocations）开关

512	关闭宏调用开关
1024	打开第一遍扫描列表开关
2048	关闭第一遍扫描列表开关
4096	打开条件汇编伪操作开关
8192	关闭条件汇编伪操作开关
16384	打开显示 MEND 伪操作开关
32768	关闭显示 MEND 伪操作开关

（2）使用说明

默认情况下，–list 汇编选项可以生成常规的汇编列表文件，该文件一般包括变量声明（variable declarations）、宏展开（macro expansions）、条件汇编伪操作（call-conditioned directive）和 MEND 伪操作（MEND directives）。汇编列表文件在第二次扫描时产生。使用 OPT 伪操作可以改变默认的汇编列表文件选项。

另外，可以通过 OPT 伪操作设置列表文件格式。如，在新的一页显示源文件的功能函数和段。

（3）示例

下面的例子显示在程序中使用 OPT 选项后 list 文件的输出结果。

源汇编文件如下所示。

```
    AREA    example,CODE,READONLY
strcopy
        LDRB    r2, [r1],#1     ;加载字节并更新字符串地址
        STRB    r2, [r0],#1     ;存储字节并更新字符串地址
        CMP     r2, #0          ;判断是否为字符串终止符
        BNE     strcopy         ;如果不是终止符，返回 strcopy 处，继续进行字符串拷贝
        MOV     pc,lr           ;子程序返回


        OPT    4


        END
```

在编译时使用了“–list”选项，汇编器输出的列表文件如下所示。

```
ARM Macro Assembler    Page 1


   1 00000000              AREA     example,CODE,READONLY
   2 00000000    strcopy
   3 00000000 E4D12001     LDRB     r2, [r1],#1;加载字节并更新字符串地址
```

```
      4 00000004 E4C02001     STRB     r2, [r0],#1;存储字节并更新字符串地址
      5 00000008 E3520000     CMP       r2, #0    ;判断是否为字符串终止符
      6 0000000C 1AFFFFFB     BNE       strcopy
                                      ;如果不是终止符，返回 strcopy，继续进行字符串拷贝
      7 00000010 E1A0F00E     MOV       pc,lr     ;子程序返回
      8 00000014
      9 00000014
      10 00000014              OPT      4



    ARM Macro Assembler    Page 2


      11 00000014
      12 00000014         END
    Command Line: --debug --liston --keep --depend="C:\Program Files\ARM\RVDS\
Examp
    les\2.2\20\windows\LDR&LDM\Directives_OPT_10\Directives_OPT_10_Debug_Direct
ives
    _OPT.s_dependency_information.txt"   --diag_style=IDE   -oDirectives_OPT_10_
Data\De
    bug\ObjectCode\Directives_OPT.o "C:\Program Files\ARM\RVDS\Examples\2.2\20\
wind
    ows\LDR&LDM\Directives_OPT.s"
```

从输出的列表文件可以看出，“OPT 4”使文件分页。

注意 如果在汇编时指定了列表文件，汇编器将信息输出到指定的列表文件中。如果没有指定列表文件名，汇编器一般将信息输出到工程文件夹下“.lst”文件中。

4．插入标题伪操作 TTL 和 SUBT

（1）语法格式

TTL 和 SUBT 为插入标题伪操作。

TTL 伪操作在列表文件的每一页开头插入一个标题。TTL 伪操作作用于其后的每一页，直到遇到新的 TTL 伪操作。

SUBT 伪操作在列表文件的每一页插入一个子标题。SUBT 伪操作作用于其后的每一页，直到遇到新的 SUBT 伪操作。

语法格式如下：

```
TTL  title
SUBT  subtitle
```

① title

插入文件列表的标题名。

② subtitle

插入文件列表的子标题名。

（2）使用说明

TTL 伪操作在列表文件的页顶部显示一个标题。如果要在列表文件的第一页显示标题，TTL 伪操作要放在源程序的第一行。

当使用 TTL 伪操作改变页标题时，新的页标题将出现在下一页的开始。

SUBT 伪操作在列表文件的页顶部显示一个子标题。如果要在列表文件的第一页显示子标题，SUBT 伪操作要放在源程序的第一行。

当使用 SUBT 伪操作改变页的子标题时，新的页子标题将出现在下一页的开始。

（3）示例

```
TTL     FirstTitle      ;
SUBT    FirstSubtitle    ;
```

10.1.5 指令集选择（Instruction Set Selection）伪操作

指示汇编器将代码编译成 32 位的 ARM 代码还是 16 位的 Thumb 代码。这类伪操作包括以下几种。

- ARM 或 CODE32 用于告诉汇编器后面的指令序列为 32 位的 ARM 指令。
- THUMB 用于告诉汇编器后面的指令是 32 位的 Thumb–2 指令还是 16 位的 Thumb 指令。
- CODE16 用于告诉汇编器后面的指令序列为 16 位的 Thumb 指令。

1．ARM 和 CODE32

（1）语法格式

ARM 伪操作指示汇编器后面的指令为 32 位的 ARM 指令。

ARM 和 CODE32 伪操作的意义相同。

当汇编器对源程序进行编译时，如果需要，将会在程序中插入空指令，以保证内存单元字对齐。

语法格式如下。

```
ARM
CODE32
```

（2）使用说明

使用在同时包含 ARM 指令和 Thumb 指令的源文件中。当需要从 Thumb 指令序列切换到 ARM 指令序列时，使用伪操作 ARM（或 CODE32）；当需要从 ARM 指令序列切换到 Thumb 指令序列时，使用 THUMB 伪操作。

注意 ARM（或 CODE32）伪操作只是指示汇编器后面的指令类型是 ARM 指令。该伪操作本身并不进行程序状态的切换，要进行状态切换，可以使用 BX 指令操作。

（3）示例

```
    AREA    Init，CODE，READONLY
    ……
    CODE32                          ;通知编译器其后的指令为 32 位的 ARM 指令
    LDR R0，=NEXT+1                 ;将跳转地址放入寄存器 R0
    BX  R0                          ;程序跳转到新的位置执行，并将处理器切换到 Thumb 工作状
态
    ……
    CODE16                          ;通知编译器其后的指令为 16 位的 Thumb 指令
    NEXT LDR R3，=0x3FF
    ……
    END                             ;程序结束
```

2．THUMB

（1）语法格式

THUMB 伪操作告诉汇编器下面的指令是 32 位 Thumb–2 指令或使用新语法的 16 位 Thumb 指令。如果需要，汇编器会在程序中插入填充位以保证内存半字节对齐。

语法格式如下。

```
    THUMB
```

（2）使用说明

如果接下来的指令使用 Thumb 语法，则使用 THUMB 伪操作指示汇编器从 ARM 状态切换到 Thumb 状态。

（3）示例

下面的程序显示了如何使用 ARM 和 THUMB 伪操作使程序从 ARM 指令切换到 Thumb 指令。

```
    AREA    ChangeState,CODE,READONLY
    ARM
    ;下面的指令在 ARM 状态下开始执行
    LDR r0,=start+1             ;取出跳转地址，设置状态标志位
    BX       r0                 ;跳转并切换程序状态

    THUMB                       ;下面的指令序列为 Thumb-2 指令
    Gsthing     PROC            ;

    B    {pc}+2                 ;#0x8002
```

```
B    {pc}+4                    ;#0x8004
```

3．CODE16

（1）语法格式

CODE16 伪指令通知编译器，其后的指令序列为 16 位的 Thumb 指令。

语法格式如下。

```
CODE16
```

（2）使用说明

若在汇编源程序中同时包含 ARM 指令和 Thumb 指令时，可用 CODE16 伪指令通知编译器其后的指令序列为 16 位的 Thumb 指令，CODE32 伪指令通知编译器其后的指令序列为 32 位的 ARM 指令。因此，在使用 ARM 指令和 Thumb 指令混合编程的代码里，可用这两条伪指令进行切换，但注意它们只通知编译器其后指令的类型，并不能对处理器进行状态的切换。

（3）示例

见 ARM（CODE32）伪操作。

10.1.6　杂项（Miscellaneous）伪操作

ARM 汇编中还有一些其他的伪操作，在汇编程序中经常会被使用，包括以下几条。

- AREA 用于定义一个代码段或数据段。
- ALIGN 用于使程序当前位置满足一定的对齐方式。
- ENTRY 用于指定程序入口点。
- END 用于指示源程序结束。
- EQU 用于定义字符名称。
- EXPORT（或 GLOBAL）用于声明符号可以被其他文件引用。
- EXPORTAS 用于向目标文件引入符号。
- IMPORT 用于告诉编译器当前符号不在本文件中。
- EXTERN 用于告诉编译器当前符号不在本文件中。
- GET（或 INCLUDE）用于将一个文件包含到当前源文件。
- INCBIN 用于将一个文件包含到当前源文件。
- RN 用于为寄存器定义名称。
- ROUT 用于定义局部变量作用范围。
- KEEP 用于将局部符号包含在目标文件的符号表中。
- NOFP 用于禁止源文件中包含浮点运算。
- REQUIRE 用于定义段之间的相互依赖关系。
- REQUIRE8 和 PRESERVE8 用于要求数据栈 8 字节对齐。

1．ALIGN

（1）语法格式

ALIGN 伪操作可通过添加填充字节的方式，使当前位置满足一定的对齐方式。

语法格式如下。

```
ALIGN{expr{, offset{, pad}}}
```

① expr

对齐表达式。表达式的值用于指定对齐方式，可能的取值为 2 的幂，如 1、2、4、8、16 等。若未指定表达式，则将当前位置对齐到下一个字的位置。

② offset

偏移量也为一个数字表达式，若使用该字段，则当前位置的对齐方式为：n*expr+偏移量。

注意 n 为汇编时变量，由编译器根据内存对齐方式决定其值。

③ pad

用作填充的字节。如果没有指定 pad，用零填充。

（2）使用说明

ALIGN 伪操作使程序代码和数据保持正确的内存对齐方式。在下面的情况中，要求特定的地址对齐方式。

① Thumb 伪指令 ADR 要求加载的地址是字对齐的，但 Thumb 代码中的标号不一定是字对齐的，这就要使用伪操作 ALIGN4 来确保程序中 Thumb 代码的地址标号是字对齐的。

② 可以使用伪操作 ALIGN 来更有效的使用 Cache。比如，ARM940T 体系结构中，Cache 是 16 字节对齐的，这时使用 ALIGN4 指定 16 字节的内存对齐方式可以充分发挥 Cache 的性能优势。

③ LDRD 和 STRD 双字传送指令要求内存 8 字节对齐。这样在 LDRD/STRD 指令所有访问的内存单元前使用 ALIGN3 实现 8 字节对齐方式。

注意 在伪操作 AREA 后使用的 ALIGN 和直接使用伪操作 ALIGN 有所不同，详见 AREA 伪操作。

（3）示例

① 通过 ALIGN 伪操作使程序中的地址标号字对齐。

```
    AREA    Example,CODE,READONLY;声明一个名为 Example 的代码段
START  LDR r0,=Sdfjk
….
    MOV    PC,LR
Sdfjk DCB 0x58                    ;定义一个字节存储空间，字对齐方式被破坏
    ALIGN                         ;声明字对齐
SUBIMOV    r1,r3                  ;其他代码
….
        MOV    PC,LR
```

② 将一个可能被 Cache 的功能段入口定义在 16 字节边界上。

```
        AREA    Cacheable，CODE，ALIGN=4
Rout1                       ;名称为 Cacheable 的代码段在 16 字节边界上对齐
        ;代码段
        MOV     pc,lr       ;字边界上对齐
        ALIGN 16            ;16 字节边界对齐
Rout2   ;代码段
…
```

③ 下面的 ALIGN 伪操作使用了 offset 偏移量。

```
AREA    OffsetExample，CODE
DCB 1
ALIGN  4，3
DCB 1
```

2．AREA

（1）语法格式

AREA 伪指令用于定义一个代码段或数据段。

ARM 程序采用分段式设计，一个 ARM 源程序至少需要一个代码段，大的程序可以包含多个代码段和数据段。关于“段”更详细的描述，可以参考相关文档。

语法格式如下。

```
AREA  sectionname{，attr}{，attr}
```

① sectionname

指定所定义段的段名。段名若以数字开头，则该段名需用“|”括起来，如：|1_test|。

注意 一些代码段具有约定的名称。如|.text|表示 C 语言编译器产生的代码段或者与 C 语言库相关的代码段。

② attr

指定代码段或数据段的属性。

在 AREA 伪操作中，各属性之间用逗号隔开。表 10.3 为各段属性及相关说明。

表 10.3 段属性及说明

段 属 性	说 明
ALIGN=expr	默认情况下，ELF 的代码段和数据段是 4 字节对齐的，expr 可以取 0～31 的数值，相应的对齐方式为 2^{expr} 字节对齐。如 expr=10，表示代码段为 1k 边界对齐。Expr 不能为 0 或 1。
ASSOC=section	指定与本段相关的 ELF 段，任何时候连接 section 段必须包含 sectionname 段
CODE	指示该段为代码段。READONLY 为默认属性

COMDEF	定义一个通用的段，该段可以包含代码或者数据。在多个源文件中同名的 COMDEF 段必须相同。如果同名的 COMDEF 段不同，连接器会报错。
COMMON	定义一个通用的数据段。该段不包括任何用户代码和数据。它被连接器自动初始化为 0。相同名称的 COMMON 段使用相同的内存单元，每个 COMMON 段的大小不必相同，连接器为其分配最大尺寸的内存。
DATA	定义数据段，默认属性为 READWRITE
NOALLOC	指定该段为虚段，并不为其在目标系统上分配内存。
NOINIT	指定本数据段不被初始化或仅初始化为 0。该操作仅为 SPACE/DCB/DCD/DCDU/DCQ/DCQ/DCW/DCWU 伪操作保留了内存单元。
READONLY	指定该段不可写，为程序代码段。
READWRITE	指定可读可写段。数据段的默认属性。

（2）使用说明

编程时使用 AREA 伪操作将程序分成多个 ELF 格式的段，段名称可以相同，这时同名的段被放在同一个 ELF 段中。ELF 段的属性根据第一个出现的 AREA 伪操作的属性设定。

一般情况下，数据段和代码段是分离的。大的程序应该被分成多个不同的代码段和数据段。一个汇编程序至少包含一个段。

（3）示例

下面伪操作定义了一个代码段，段名为 Init，属性为只读。

```
AREA    Init, CODE, READONLY
; code
```

3．END

（1）语法格式

END 伪操作用于通知编译器已经到了源程序的结尾。

语法格式如下。

```
END
```

（2）使用说明

每一个汇编源文件必须以 END 结束。

如果汇编文件通过伪操作 GET 指定了一个“父文件（parent file）”，当汇编器遇到 END 伪操作时将返回到“父文件”继续汇编。

（3）示例

使用 END 伪操作指定应用程序的结尾。

```
AREA    Init，CODE，READONLY
……
END             ;
```

4．ENTRY

（1）语法格式

ENTRY 伪操作用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个 ENTRY（也可以有多个，当有多个 ENTRY 时，程序的真正入口点由链接器指定），但在一个源文件里最多只能有一个 ENTRY（可以没有）。

语法格式如下。

```
ENTRY
```

（2）使用说明

在一个完整的汇编程序中至少要有一个 ENTRY，如果在程序连接时没有发现 ENTRY 伪操作，连接器将产生警告信息。

在一个源文件里最多只能有一个 ENTRY，如果多个 ENTRY 同时出现在源文件中，汇编时将产生错误信息。

（3）示例

使用伪操作 ENTRY 指定程序入口点。

```
AREA    Init，CODE，READONLY
ENTRY                    ;指定应用程序的入口点
……
```

5．EQU

（1）语法格式

EQU 伪操作用于为程序中的常量、标号等定义一个等效的字符名称，类似于 C 语言中的＃define。其中 EQU 可用“*”代替。

语法格式如下。

```
name  EQU  expr{,type}
```

① name

EQU 伪指令定义的字符名称。

② expr

32 位表达式。其值为基于寄存器的地址值、程序中的标号、32 位的地址常量或 32 位的常量。

③ type

指定数据类型。为一个可选项。

当表达式 expr 为 32 位的常量时，可以指定表达式的数据类型，可以有以下几种类型：CODE16、CODE32、ARM、THUMB 和 DATA。

当定义的名称（name）被声明为可被其他文件引用（exported）时，在目标文件的符号表中将包含名称（name）的数据类型。这些信息将会被连接器使用。

（2）使用说明

EQU 类似于 C 语言中的＃define 操作。

（3）示例

定义标号 Test 的值为 50，定义 Addr 的值为 0x55。

```
Test EQU 50                ;定义标号 Test 的值为 50
Addr EQU 0x55,CODE32       ;定义 Addr 的值为 0x55，且该处为 32 位的 ARM 指令。
```

6．EXPORT（或 GLOBAL）

（1）语法格式

EXPORT 伪操作用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。EXPORT 可用 GLOBAL 代替。标号在程序中区分大小写。

语法格式如下。

```
EXPORT{symbol}{[WEAK，attr]}
```

① symbol

被声明的符号名称。名称区分大小写。如果 symbol 被忽略，所有符号被定义为可以被其他文件引用属性。

② [WEAK]

[WEAK]选项声明其他的同名标号优先于该标号被引用。

③ [attr]

符号属性。用于定义所定义的符号对其他文件的“可见性（visibility）”。默认情况下，被定义为全局的（global）的符号对其他文件是“可见的”，也就是说可以被其他文件引用。而定义为本地（local）的符号对其他文件是“不可见的”，即不可被其他文件引用。

attr 可以是下面一些属性。

- DYNAMIC：符号可以被其他文件引用，且可以在其他文件中被重新定义。
- HIDDEN：符号不能其他组件引用。
- PROTECTED：符号可以被其他文件引用，但不可重新定义。

（2）使用说明

EXPORT 声明的变量可以被其他文件访问。

（3）示例

声明一个可全局引用的标号 Stest。

```
AREA    Init,CODE，READONLY
EXPORT        Stest              ;声明一个可全局引用的标号 Stest
……
END
```

7．EXPORTAS

（1）语法格式

EXPORTAS 用于修改已被编译的目标文件中的符号。

语法格式如下。

```
EXPORTAS  symbol1，symbol2
```

① symbol1

源文件中的符号名。symbol1 必须在源文件中已被定义。它可以是段名、标号或常量。

② symbol2

目标文件中的符号名。它将取代目标文件中的 symbol1 符号。该符号名称区分大小写。

（2）使用说明

用于修改目标文件中的符号定义。

（3）示例

```
AREA data1, DATA            ;定义新的数据段 data1
AREA data2, DATA            ;定义新的数据段 data2
 EXPORTAS data2, data1      ;data2 中定义的符号将会出现在 data1 的符号表中
one EQU  2
    EXPORTAS one, two
    EXPORT one              ;符号 two 将在目标文件中以 “2” 的形式出现
```

8．EXTERN

（1）语法格式

EXTERN 伪操作用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。

标号在程序中区分大小写。

语法格式如下。

```
EXTERN  symbol{[WEAK，attr]}
```

① symbol

要引用的符号名称。该名称区分大小写。

② [WEAK]

[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为 0，若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。

③ [attr]

符号属性。用于定义所定义的符号对其他文件的“可见性（visibility）”。默认情况下，被定义为全局的（global）的符号对其他文件是“可见的”，也就是说，可以被其他文件引用。而定义为本地（local）的符号对其他文件是“不可见的”，即不可被其他文件引用。

[attr]可以是下面一些属性。

- DYNAMIC：符号可以被其他文件引用，且可以在其他文件中被重新定义。
- HIDDEN：符号不能其他文件引用。
- PROTECTED：符号可以被其他文件引用，但不可重新定义。

（2）使用说明

当源文件的的符号用 EXTERN 声明后，该符号在连接时被解析。

（3）示例

① 通知编译器当前文件要引用标号 Main，但 Main 在其他源文件中。

```
AREA    Init，CODE，READONLY
EXTERN       Main    ;通知编译器当前文件要引用标号 Main，但 Main 在其他源文件中定义
……
END
```

② 下面的程序用于检测 C++库是否被连接，并根据检测结果，执行指令跳转。

```
AREA    Example, CODE, READONLY
EXTERN  __CPP_INITIALIZE[WEAK]      ;如果 C++ 库被连接
                                    ;得到__CPP_INITIALIZE 函数的入口地址
 LDR     r0,=__CPP_INITIALIZE       ;如果没有被连接，地址为 0
 CMP     r0,#0                      ;如果为 0.
 BEQ     nocplusplus                ;跳转到相应函数
```

9．GET（或 INCLUDE）

（1）语法格式

GET 伪操作用于将一个源文件包含到当前的源文件中，并将被包含的源文件在当前位置进行汇编处理。可以使用 INCLUDE 代替 GET。

语法格式如下。

```
GET  filename
```

其中，filename 是被包含的文件名称。ARM 汇编器接受的路径名称可以是 UNIX 或 MS-DOS 的路径格式。

（2）使用说明

汇编程序中常用的方法是在某源文件中定义一些宏指令，用 EQU 定义常量的符号名称，用 MAP 和 FIELD 定义结构化的数据类型，然后用 GET 伪指令将这个源文件包含到其他的源文件中。使用方法与 C 语言中的“include”相似。

GET 伪操作只能用于包含源文件，包含目标文件需要使用 INCBIN 伪操作。

（3）示例

通知编译器当前源文件包含源文件 a1.s 和源文件 C：\a2.s。

```
AREA    Init，CODE，READONLY
GET  a1.s                    ;通知编译器当前源文件包含 a1.s
```

```
GE  T    C: \a2.s            ;通知编译器当前源文件包含 C: \ a2.s
……
END
```

10．IMPORT

（1）语法格式

IMPORT 伪操作用于通知编译器要使用的标号在其他的源文件中定义。

> **注意** IMPORT 和 EXTERN 用法相似，IMPORT 声明的符号无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。EXTERN 声明的符号，如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。

标号在程序中区分大小写。

语法格式如下。

```
IMPORT  symbol{[WEAK，attr]}
```

① symbol

被声明的符号名称。名称区分大小写。如果 symbol 被忽略，所有符号被定义为可以被其他文件引用属性。

② [WEAK]

[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器不给出错误信息，在多数情况下将该标号置为 0，若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。

③ [attr]

符号属性。用于定义所定义的符号对其他文件的“可见性（visibility）”。默认情况下，被定义为全局的（global）的符号对其他文件是“可见的”，也就是说，可以被其他文件引用。定义为本地（local）的符号对其他文件是“不可见的”，即不可被其他文件引用。

[attr]可以是下面一些属性。

- DYNAMIC：符号可以被其他文件引用，且可以在其他文件中被重新定义。
- HIDDEN：符号不能其他组件引用。
- PROTECTED：符号可以被其他文件引用，但不可重新定义。

（2）使用说明

当源文件的的符号用 IMPORT 声明后，该符号在连接时被解析。

（3）示例

参见 EXTERN 伪操作。

11．INCBIN

（1）语法格式

INCBIN 伪操作用于将一个目标文件或数据文件包含到当前的源文件中，被包含的文件不作任何变动地存放在当前文件中，编译器从其后开始继续处理。

语法格式如下。

```
INCBIN  filename
```

其中 filename 指定将要包含进当前源文件的文件名。汇编器接受的路径名称可以是 UNIX 或 MS-DOS 的路径格式。

（2）使用说明

使用 INCBIN 可以包含任何格式的文件，如二进制文件、字符文件等。汇编器对此文件内容不做任何修改。

（3）示例

通知编译器当前源文件包含文件 a1.dat 和 C:\a2.txt。

```
AREA    Init，CODE，READONLY
INCBIN       a1.dat           ;通知编译器当前源文件包含文件 a1.dat
INCBIN  C: \a2.txt            ;通知编译器当前源文件包含文件 C: \a2.txt
……
END
```

12．KEEP

（1）语法格式

KEEP 伪操作将本地符号包含进目标文件的符号表中。

语法格式如下。

```
KEEP  {symbol}
```

其中 symbol 是将要包含进符号表的本地变量，如果程序中没有指定 symbol，则所有本地变量将会被包含进符号表。

（2）使用说明

默认情况下，汇编器只将用“exported”声明的变量或被重定位的变量包含进符号表。使用 KEEP 伪操作可将需要的本地变量包含进符号表，以方便调试。

注意 KEEP 不能将寄存器相关的符号包含进符号表。请参加 MAP 伪操作。

（3）示例

下面的例子将 label 包含进符号表。

```
label   ADC r2,r3,r4
        KEEP    label           ;指示汇编器将 label 包含进符号表
        ADD r2,r2,r5
```

13．NOFP

（1）语法格式

NOFP 伪操作用于告诉汇编器当前源文件中不包含浮点运算指令。

语法格式如下。

```
NOFP
```

（2）使用说明

在软硬件件均不支持浮点运算的情况下可以使用 NOFP 伪操作确保编译出的源文件不包含浮点运算指令。

如果在 NOFP 伪操作之后出现浮点运算指令编译器将报告“无法识别操作码错误（Unknown opcode error）”。

如果 NOFP 伪操作在浮点运算指令之后定义，汇编器将产生“浮点运算汇编后错误（Too Late to ban floating poing instruction）”。

14．REQUIRE

（1）语法格式

REQUIRE 伪操作用于定义段之间的相互依赖关系。

语法格式如下。

```
REQUIRE  label
```

其中 label 为所要指定的标号名称。

（2）使用说明

REQUIRE 伪操作用于定义相关段的包含关系。如果一个含有 REQUIRE 伪操作的段被连接进目标文件，那么在该文件中由 REQUIRE 指定的文件也将被连接。

15．REQUIRE8（或 PRESERVE8）

（1）语法格式

REQUIRE8 伪操作指定当前文件堆栈要求 8 字节对齐。它将传递 REQ8 连接选项到 ARM 连接器。

PRESERVE8 伪操作指定当前文件堆栈要求 8 字节对齐。它将传递 RPES8 连接选项到 ARM 连接器。

连接器保证要求 8 字节堆栈对齐的代码相互调用。

语法格式如下。

```
REQUIRE8 {bool}
PRESERVE8 {bool}
```

其中 bool 的取值为{TRUE}或{FALSE}

（2）使用说明

在 ARMv6 版本之前的体系结构中使用 LDRD 或 STRD 指令，要求存取的数据 8 字节对齐。为了在程序中正确使用 LDRD 或 STRD 指令，要使用 REQUIRE8（或 PRESERVE8）伪操作。

注意 在 ARMv6 体系中，LDRD 和 STRD 指令不再要求数据 8 字节对齐而是 4 字节对齐。

如果程序中包含 LDRD 或 STRD 指令而没有使用伪操作 REQUIRE8（或 PRESERVE8）声明，汇编器将给出警告。

ARM 公司建议在包含 LDRD 或 STRD 的代码中，显性地定义 REQUIRE8（或 PRESERVE8）。可以使用“armasm --diag_waring 1546”汇编命令，让汇编器给出堆栈 8 字节字符对齐警告。警告的形式如下。

```
"test.s", line 37: Warning: A1546W: Stack pointer update potentially
 breaks 8 byte stack alignment
 37 00000044  STMFD  sp!,{r2,r3,lr}
```

（3）示例

下面几个例子给出了 REQUIRE8 伪操作的标准用法。

```
REQUIRE8
REQUIRE8    {TRUE}     ;
REQUIRE8    {FALSE}    ;
PRESERVE8   {TRUE}     ;
PRESERVE8   {FALSE}    ;
```

16．RN

（1）语法格式

RN 伪操作用于给一个寄存器定义一个别名。采用这种方式可以方便程序员记忆该寄存器的功能。

语法格式如下。

```
name  RN  expr
```

① name

为寄存器起的别名。所起的名字不能和预定义的寄存器和协处理器名字相同。

② expr

寄存器编号。取值范围为 0～15。

（2）使用说明

采用给寄存器起别名的方式可以方便程序员记忆该寄存器的功能。

（3）示例

```
regname    RN  11  ;将寄存器 11 命名为 regname
sqr4       RN  r6  ;将寄存器 6 定义为 spr4
```

17．ROUT

（1）语法格式

ROUT 伪操作用于给一个局部变量定义作用范围。在程序中未使用该伪指令时，局部变量的作用范围为所在的 AREA，而使用 ROUT 后，局部变量的作为范围为当前 ROUT 和下一

个 ROUT 之间。

语法格式如下。

```
{name}  ROUT
```

其中 name 是所定义的局部变量的名称。

（2）使用说明

将一个变量定义为局部变量可以避免程序中误操作。

（3）示例

```
            ; code
routineA    ROUT                ;定义局部标号的有效范围，范围名称为 routineA
            ; code
3routineA   ; code              ;routineA 范围内的局部标号 3
            ; code
            BEQ     %4routineA  ;若条件成立，跳转到 routineA 范围内的局部号 4
            ; code
            BGE     %3          ;若条件成立，跳转到 routineA 范围内的局部标号 3
            ; code
4routineA   ; code              ;范围 A 内的局部标号 4
            ; code
otherstuff  ROUT                ;定义新的局部标号有效范围
```

10.1.7　结构描述伪操作

栈中数据结构描述伪操作主要用于调试，所以将其放在伪操作的最后一部分介绍。感兴趣的读者可参考 ARM 相关文档，获得更详细的信息。

在程序中使用结构描述伪操作可以达到以下目的。

- 允许用 armlink -callgraph 选项计算汇编程序函数的栈使用量。
- 帮助避免函数构造中的错误，特别是正在修改现有的代码时，结构描述伪操作将会给编程者带来很大帮助。
- 允许汇编器对函数构造中的错误发出警告。
- 在调试时启用函数调用的回溯。
- 允许调试程序剖析（Profile）汇编函数。

在调试程序时，需要剖析（Porfile）函数，但不希望使用结构描述伪操作，可以使用以下办法。

- 使用 FUNCTION 和 ENDFUNC 伪操作，或者使用 PROC 和 ENOP 伪操作。
- 可以只为需要剖析的函数使用 FUNCTION 和 ENDFUNC 伪操作。

1．FRAME　ADDRESS

（1）语法格式

FRAME ADDRESS 伪操作说明如何为后面的指令计算规范框架地址。只能在含有 FUNCTION 和 ENDFUNC 伪操作或含有 PROC 和 ENDP 伪操作的函数内使用它。

语法格式如下。

```
FRAME ADDRESS reg,[offset]
```

① reg

规范结构地址寄存器，如果函数没有使用独立的结构指针，则该寄存器为栈指针寄存器 SP。

② offset

规范结构地址从 reg 算起的偏移量。如果 offset 等于 0，则被忽略。

（2）使用说明

如果代码改变了规范结构地址的基址寄存器，或者如果代码改变了规范结构地址对该寄存器的偏移量，就要使用 FRAME ADDRESS 伪操作定义新的地址。必须在改变规范结构地址的计算方式的指令之后，立即使用 FRAME ADDRESS 伪操作。

注意

如果代码只使用了单个指令来保存寄存器和改变栈指针，可以使用 FRAME PUSH 伪操作来代替 FRAME ADDRESS 和 FRAME SAVE 伪操作。

如果代码使用了单个指令来装载寄存器和改变栈指针，可以使用 FRAME POP 来代替 FRAME ADDRESS 和 FRAME RESTORE 伪操作。

（3）示例

```
_fn FUNCTION            ;CFA (规范结构地址 Canonical Frame Address) 值等于 SP 的值
        PUSH {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB sp,sp,#4        ;改变了 CFA 的偏移量
        FRAME ADDRESS sp,24 ;修正该地址
        ADD fp,sp,#20
        FRAME ADDRESS fp,4  ;将 fp 作为新基址寄存器
```

2. FRAME POP

（1）编码格式

当被调用代码重新装载寄存器时，使用 FRAME POP 伪操作来通知汇编程序。只能在包含 FUNCTION 和 ENDFUNC 伪操作或在 PROC 和 ENDP 伪操作的函数内使用它。

在函数的最后一条指令后不需要使用。

语法格式如下。

```
FRAME POP {reglist} 或
FRAME POP n
```

① reglist

在函数入口处保存数据的寄存器列表。该列表至少要包含一个寄存器。

② *n*

堆栈指针移动的字节数。

（2）使用说明

FRAME POP 等价于 FRAME ADDRESS 和 FRAME RESTORE 伪操作。当单个指令装载寄存器和改变栈指针时，可以使用它。

必须在加载寄存器或改变栈指针的指令后立即使用 FRAME POP 伪操作。

汇编程序根据以下假定计算新的规范结构地址的偏移量。

① 出栈的每个 ARM 寄存器占用栈内的 4 个字节。

② 出栈的每个 FPA 浮点寄存器占用栈内的 12 个字节。

③ 出栈的每个 VFP 单精度寄存器占用栈内的 4 个字节，对每个列表另加上一个附加的 4 字节字。

3．FRAME　PUSH

（1）语法格式

当被调用函数保存寄存器时（通常在函数入口点处），使用 FRAME PUSH 伪操作来通知汇编程序。只能在包含 FUNCTION 和 ENDFUNC 伪操作或在 PROC 和 ENDP 伪操作的函数内使用它。

语法格式如下。

```
FRAME  PUSH  {reglist}  或，
FRAME  PUSH  n
```

① reglist

是一个在规范结构地址后连续保存的寄存器列表。列表中至少包含一个寄存器。

② *n*

栈指针移动的字节数。

（2）使用说明

FRAME PUSH 等价于 FRAME ADDRESS 和 FRAME SAVE 伪操作。当单个指令装载寄存器或改变栈指针时，可以使用它。

必须在装载寄存器或改变栈指针的指令后立即使用 FRAME PUSH 伪操作。

汇编程序根据以下假定计算新的规范框架地址的偏移量。

① 入栈的每个 ARM 寄存器占用栈内的 4 个字节。

② 入栈的每个 FPA 浮点寄存器占用栈内的 12 个字节。

③ 入栈的每个 VFP 单精度寄存器占用栈内的 4 个字节，对每个列表另加上一个附加的 4 字节字。

（3）示例

```
p  PROC                    ;规范框架地址等于 sp + 0
   EXPORT p
   PUSH {r4-r6,lr}         ;寄存器 r4，r5，r6 和 lr 入栈，sp 指向相关的规范框架地址
   FRAME PUSH {r4-r6,lr}
```

```
;等价于下面的伪操作:
;FRAME  ADDRESS  sp,16   ;16 bytes in {r4-r6,lr}
;FRAME SAVE {r4-r6,lr},-16
```

4．FRAME REGISTER

（1）编码格式

使用 FRAME REGISTER 伪操作保存寄存器中存放的函数自变量位置的一个记录。只能在使用 FUNCTION 和 ENDFUNC 伪操作或在 PROC 和 ENDP 伪操作的函数内使用它。

语法格式如下。

```
FRAME  REGISTER  reg1，reg2
```

① reg1

存放函数入口点自变量的寄存器。

② reg2

指定用于保存参数值的寄存器。

（2）使用说明

在进入函数时，使用寄存器保存函数自变量。此时使用该伪操作通知编译程序。

5．FRAME RESTORE

（1）编码格式

使用 FRAME RESTORE 伪操作通知汇编程序指定的寄存器的内容已被恢复为进入函数时存放的值。只能在使用 FUNCTION 和 ENDFUNC 伪操作或在 PROC 和 ENDP 伪操作的函数内使用它。

语法格式如下。

```
FRAME  RESTORE {reglist}
```

其中，reglist 为已恢复其内容的寄存器列表。列表中至少包含一个寄存器。

注意 reglist 可以包含整数寄存器或浮点寄存器，但两者不能同时包含。

（2）使用说明

在被调用寄存器从栈中重新加载寄存器后，立即使用 FRAME RESTORE 伪操作。在函数的最后一条指令后不需要使用该指令。

6．FRAME RETURN ADDRESS

（1）语法格式

FRAME RETURN ADDRESS 伪操作为使用非 r14 的寄存器的函数提供返回地址。只能在使用 FUNCTION 和 ENDFUNC 伪操作或与 PROC 和 ENDP 伪操作的函数内使用它。

注意 使用非 r14 寄存器作为其返回地址的任何函数与 ATPCS 不兼容。这种函数不能被导出。

语法格式如下。

```
FRAME RETURN ADDRESS reg
```

其中：reg 是用作返回地址的寄存器。

（2）使用说明

在不使用 r14 作为其返回地址的任何函数中使用 FRAME RETURN ADDRESS 伪操作，否则，调试程序不能回溯该函数。

在引入该函数的 FUNCTION 或 PROC 伪操作后立即使用 FRAME RETURN ADDRESS 伪操作。

7．FRAME SAVE

（1）语法格式

FRAME SAVE 伪操作描述所保存的寄存器内容相对于规范框架地址的位置。只能在包含 FUNCTION 和 ENDFUNC 伪操作或与 PROC 和 ENDP 伪操作的函数内使用它。

语法格式如下。

```
FRAME SAVE {reglist}, offset
```

① reglist

是从规范框架地址偏移 offset 处开始连续保存的一个寄存器列表。列表中必须至少包含一个寄存器。

② offset

偏移地址

（2）使用说明

在被调用函数将寄存器存储到栈内以后，立即使用 FRAME SAVE 伪操作。

reglist 可以包含回溯不需要的寄存器。汇编程序确定需要在 DWARF 调用框架信息中记录哪些寄存器。

注意 如果代码使用单个指令来保存寄存器和改变堆栈指针，可以使用 FRAME PUSH 来代替 FRAME SAVE 和 FRAME ADDRESS 伪操作。

8．FRAME STATE REMEMBER

（1）编码格式

FRAME STATE REMEMBER 伪操作用于保存有关如何计算规范框架地址以及已保存的寄存器值的位置的当前信息。只能在包含 FUNCTION 和 ENDFUNC 伪操作或 PROC 和 ENDP 伪操作的函数内使用它。

语法格式如下。

```
FRAME STATE REMEMBER
```

（2）使用说明

在一个内联的退出序列过程中，会改变规范框架地址的计算信息以及已保存的寄存器值的位置信息。在退出序列之后，另一个跳转可以继续像以前一样使用相同的信息。使用 FRAME STATE REMEMBER 来保存这些信息，使用 FRAME STATE RESTORE 来恢复它。

可以嵌套这些命令。每个 FRAME STATE RESTORE 伪操作必须有对应的 FRAME STATE REMEMBER 伪操作。

（3）示例

```
        ; ……
        FRAME STATE REMEMBER
            ; 内嵌退出之前保存帧状态
        POP {r4-r6,pc}
            ; 不再需要 FRAME POP 伪操作
        FRAME STATE RESTORE
            ; 恢复帧状态
exitB ; 恢复之后可以开始 exitB 的代码
        POP {r4-r6,pc}
        ENDP
```

9．FRAME STATE RESTORE

（1）语法格式如下

FRAME STATE RESTORE 伪操作用于恢复有关如何计算信息规范框架地址和已保存的寄存器值的位置信息。只能在包含 FUNCTION 和 ENDFUNC 伪操作或在 PROC 和 ENDP 伪操作的函数内使用它。

语法格式如下。

```
FRAME STATE RESTORE
```

（2）使用说明

参见 FRAME REMEMEBER 伪操作。

10．FRAME UNWIND ON

（1）语法格式

FRAME UNWIND ON 伪操作指示汇编器为后面的函数生成 UNWIND 列表。

语法格式如下。

```
FRAME UNWIND ON
```

（2）使用说明

可以在函数体外使用该伪操作。这样，汇编器将为接下来的所有函数产生 unwind 列表，

直到出现 UNWIND OFF 伪操作。

11．FRAME UNWIND OFF

（1）语法格式

FRAME UNWIND ON 伪操作指示汇编器为后面的函数生成 NOUNWIND 列表。

语法格式如下。

```
FRAME UNWIND ON
```

（2）使用说明

可以在函数体外使用该伪操作。这样，汇编器将为接下来的所有函数产生 NOUNWIND 列表，直到出现 UNWIND ON 伪操作。

12．FUNCTION 或 PROC

（1）语法格式

FUNCTION 伪操作标记一个兼容 AAPCS 的函数开始。PROC 和 FUNCTION 作用相同。

语法格式如下。

```
label FUNCTION [{Reglist1} [, {Reglist2}]]
```

① Reglist1

Reglist1 是一个可选的由被调用函数保存的 ARM 寄存器列表。如果 reglist1 不出现并且调试程序检查寄存器使用情况，则将假定函数符合 AAPCS 规则。

② Reglist2

Reglist2 是一个可选的由被调用函数保存的 VFP 寄存器列表。

（2）使用说明

使用 FUNCTION 来标记函数的开始。在为 ELF 生成 DWARF 调用框架信息时汇编程序使用 FUNCTION 来标识一个函数的开始。FUNCTION 将规范框架地址设置为 sp 并将框架状态栈清空。

每个 FUNCTION 伪操作必须有一个相匹配的 ENDFUNC 伪操作，也就是说 FUNCTION 和 ENDFUNC 必须成对出现。不能嵌套 FUNCTION/ENDFUNC 对，并且它们不能包含 PROC 或 ENDP 命令。

如果正在使用自己的程序调用标准，可以使用可选的 reglist 参数来将有关此备选程序调用标准的信息通知调试程序，并非所有调试程序都支持此功能。详细信息请参阅所用调试器的程序。

（3）示例

```
     ALIGN                          ;确保对齐
dadd FUNCTION
     EXPORT dadd
     PUSH {r4-r6,lr}                ;自动保证字对齐
     FRAME PUSH {r4-r6,lr}
```

```
        ;子函数体
        ; ……
        POP {r4-r6,pc}
        ENDFUNC
func6 PROC {r4-r8,r12},{D1-D3}   ;不遵循 AAPCS 标准的函数
...
        ENDP
```

13．ENDFUNC 或 ENDP

ENDFUNC 伪操作标记一个遵循 AAPCS 规则的函数体结束。ENDP 和 ENDFUNC 作用相同。

详见 FUNCTION 或 PROC 伪操作部分。

10.2 ARM 汇编器所支持的伪指令

ARM 汇编器支持 ARM 伪指令，这些伪指令在汇编阶段被翻译成 ARM 或者 Thumb（或 Thumb-2）指令（或指令序列）。ARM 伪指令包含 ADR、ADRL、MOV32 和 LDR。

- ADR 伪指令装载程序相关（program-relative）或寄存器相关（register-relative）地址（小范围地址）到寄存器。
- ADRL 伪指令装载程序相关（program-relative）或寄存器相关（register-relative）地址（中等范围地址）到寄存器。
- MOV32 装载 32 位常数或地址到寄存器（ARMv6T2 体系结构及以上版本支持）。
- LDR 装载 32 位常数或地址到寄存器（所有 ARM 版本均支持）。

10.2.1 ADR 伪指令

（1）语法格式

ADR 为小范围地址读取伪指令。ADR 伪指令将基于 PC 相对偏移地址或基于寄存器相对偏移地址值读取到寄存器中，当地址值是字节对齐时，取值范围为−255～255，当地址值是字对齐时，取值范围−1020～1020。当地址值是 16 字节对齐时其取值范围更大。

语法格式如下。

```
ADR{cond}{.W}  register，label
```

① cond

可选的指令执行条件。

② .W

可选项。指定指令宽度（Thumb-2 指令集支持）。

③ register

目标寄存器。

④ label

基于 PC 或具有寄存器的表达式。

（2）使用说明

ADR 被汇编器编译成一条指令。汇编器通常使用 ADD 或 SUB 指令来实现伪操作的地址装载功能。如果不能用一条指令来实现 ADR 伪指令的功能，汇编器将报告错误。

（3）示例

```
        LDR     r4,=data+4*n        ;n 是汇编时产生的变量
        ; code
        MOV     pc,lr
data    DCD     value0
        ; n-1 条 DCD 伪操作
        DCD     valuen              ;所要装载入 r4 的值
        ;更多 DCD 伪操作
```

10.2.2 ADRL 伪指令

（1）语法格式

ADRL 为中等范围地址读取伪指令。ADRL 伪指令将基于 PC 相对偏移的地址或基于寄存器相对偏移的地址值读取到寄存器中，当地址值是字节对齐时，取值范围为−64KB～64KB，当地址值是字对齐时，取值范围为−256KB～256KB。当地址值是 16 字节对齐时，其取值范围更大。在 32-bit 的 Thumb-2 指令中，地址取值范围到达±1MB。

注意 ADRL 只能用在 ARM 汇编或 Thumb－2 汇编中，Thumb 汇编器不支持 ADRL 伪指令。

语法格式如下。

```
ADRL{cond}  register，label
```

① cond

可选的指令执行条件。

② register

目标寄存器。

③ label

基于 PC 或具体寄存器的表达式。

（2）使用说明

ADRL 伪指令与 ADR 伪指令相似，用于将基于 PC 相对偏移的地址或基于寄存器相对偏移的地址值读取到寄存器中。所不同的是 ADRL 伪指令比 ADR 伪指令可以读取更大范围的地址。这是因为在编译阶段，ADRL 伪指令被编译器换成两条指令。即使一条指令可以完成该操作，编译器也将产生两条指令，其中一条为多余指令。如果汇编器不能在两条指令内完成操作，将报告错误，中止编译。

注意 如果使用 ADRL 伪指令装载标号地址，那么被装载的标号必须和伪指令 ADRL 在同一个段中。

10.2.3 MOV32 伪指令

（1）语法格式

MOV32 伪指令装载一个 32 位常数或地址到寄存器。

与 ADR 和 ADRL 指令不同，MOV32 伪指令装载的地址是位置相关地址。

语法格式如下。

```
MOV32{cond}  register，expr
```

① cond

可选的指令执行条件。

② register

目标寄存器。

③ expr

表达式，可以是下面几种形式。

- symbol：程序中定义的标号地址。
- constant：任意 32-bit 的常数。
- symbol+constant：地址标号加 32-bit 常数。

（2）使用说明

一般情况下汇编器将一条 MOV32 伪指令翻译成一对 MOV 和 MOVT 指令。这样任何 32-bit 常量都可以被装载到寄存器。

注意 通过这种方法装载的程序地址是在程序链接时由链接器确定的，是位置相关地址。

10.2.4 LDR 伪指令

（1）语法格式

LDR 伪指令装载一个 32-bit 常数和一个地址到寄存器。

语法格式如下。

```
LDR{cond}{.W}  register，=[expr|label-expr]
```

① cond

可选的指令执行条件。

② .W

可选项。指定指令宽度（Thumb-2 指令集支持）。

③ register

目标寄存器。

④ expr

32 位常量表达式。汇编器根据 expr 的取值情况，对 LDR 伪指令做如下处理。

- 当 expr 表示的地址值没有超过 MOV 或 MVN 指令的地址取值范围时，汇编器用一对 MOV 和 MVN 指令代替 LDR 指令。
- 当 expr 表示的指令地址值超过了 MOV 或 MVN 指令的地址范围时，汇编器将常数放入数据缓存池，同时用一条基于 PC 的 LDR 指令读取该常数。

⑤ label-expr

一个程序相关或声明为外部的表达式。汇编器将 label-expr 表达式的值放入数据缓存池，使用一条程序相关 LDR 指令将该值取出放入寄存器。

当 label-expr 为声明为外部的表示式时，汇编器将在目标文件中插入链接重定位伪操作，由链接器在链接时生成该地址。

（2）使用说明

当要装载的常量超出了 MOV 或 MVN 指令的范围时，使用 LDR 指令。

由 LDR 指令装载的地址是绝对地址，即 PC 相关地址。

当要装载的数据不能由 MOV 或 MVN 指令直接装载时，该值要先放入数据缓存池，此时 LDR 伪指令处的 PC 值到数据缓存池中目标数据所在地址的偏移量有一定限制。ARM 或 32-bit 的 Thumb-2 指令中该范围是±4KB，Thumb 或 16-bit 的 Thumb-2 指令中为 0～1KB。

（3）示例

① 将常数 0xff0 读到 r1 中。

```
LDR r3,=0xff0 ;
```

相当于下面的 ARM 指令：

```
MOV r3,#0xff0
```

② 将常数 0xfff 读到 r1 中。

```
LDR r1,=0xfff ;
```

相当于下面的 ARM 指令：

```
LDR r1,[pc,offset_to_litpool]
...
litpool DCD 0xfff
```

③ 将 place 标号地址读入 r1 中。

```
LDR r2,=place ;
```

相当于下面的 ARM 指令：

```
LDR r2,[pc,offset_to_litpool]
...
litpool DCD place
```

10.3 汇编语言文件格式

10.3.1 ARM 汇编语言语句格式

ARM（Thumb）汇编语法语句格式如下所示。

```
{symbol}{instruction|directive|pseudo-instruction}{;comment}
```

① symbol

程序符号。通常为地址标号（label）。在指令和伪指令中通常为标号；在一些伪操作中符号可能是变量或常数。详见 ARM 伪操作一节。

在书写中，符号必须从一行的行头开始，前面不能包含空格或制表符 tab。

② instruction

ARM 或 Thumb 指令。

③ directive

伪操作。详见 ARM 伪操作一节。

④ pseudo-instruction

ARM 伪指令。详见 ARM 伪指令一节。

⑤ comment

语句注释。注释以分号（;）开头，注释的结尾即为一行的结尾。为了程序清晰易读，注释也可以单独占用一行。汇编器在对程序进行汇编时忽略注释。

在汇编语言程序设计中，每一条指令的助记符可以全部用大写或全部用小写，但不允许在一条指令中大、小写混用。

同时，如果一条语句太长，可将该长语句分为若干行来书写，在行的末尾用“\”表示下一行与本行为同一条语句。

10.3.2 ARM 汇编语言中的符号

在汇编语言程序设计中，经常使用各种符号代替地址（addresses）、变量（variables）和常量（constants）等，以增加程序的灵活性和可读性。尽管符号的命名由编程者决定，但并不是任意的，必须遵循以下的约定。

（1）符号区分大小写，同名的大、小写符号会被编译器认为是两个不同的符号。

（2）符号在其作用范围内必须惟一。

（3）自定义的符号名不能与系统的保留字相同。其中保留字包括系统内部变量（built in variable）和系统预定义（predefined symbol）的符号。

（4）符号名不应与指令或伪指令同名。如果要使用和指令或伪指令同名的符号要用双斜杠“||”将其括起来，如“|ASSERT|”。

注意 虽然符号被双斜杠括起来，但双斜杠并非符号名的一部分。

（5）局部标号以数字开头，其他的符号都不能以数字开头。

1．变量（variable）

程序中的变量是指其值在程序的运行过程中可以改变的量。ARM（Thumb）汇编程序所支持的变量有三种。

- 数字变量（numeric）。
- 逻辑变量（logical）。
- 字符串变量（string）。

数字变量用于在程序的运行中保存数字值，但注意数字值的大小不应超出数字变量所能表示的范围。

逻辑变量用于在程序的运行中保存逻辑值，逻辑值只有两种取值情况：真（{TURE}）和假（{FALSE}）。

字符串变量用于在程序的运行中保存一个字符串，注意字符串的长度不应超出字符串变量所能表示的范围。

在 ARM（Thumb）汇编语言程序设计中，可使用 GBLA、GBLL、GBLS 伪指令声明全局变量，使用 LCLA、LCLL、LCLS 伪指令声明局部变量，可使用 SETA、SETL 和 SETS 对其进行初始化。

2．常量（constants）

程序中的常量是指其值在程序的运行过程中不能被改变的量。ARM（Thumb）汇编程序所支持的常量有数字常量、逻辑常量和字符串常量。

数字常量一般为 32 位的整数，当作为无符号数时，其取值范围为 $0\sim2^{32}-1$，当作为有符号数时，其取值范围为$-2^{31}\sim2^{31}-1$。汇编器认为−n 和 $2^{32}-n$ 是相等的。对于关系操作，如比较两个数的大小，汇编器将其操作数看作无符号的数，也就是说“0>−1”，对汇编器来说取值为“假（{FLASE}）”。

逻辑常量只有两种取值情况，真或假。

字符串常量为一个固定的字符串，一般用于程序运行时的信息提示。

3．程序中的变量代换

汇编语言中的变量可以作为作为一整行出现在汇编程序中，也可以作为行的一部分使用。

如果在数字变量前面有一个代换操作符“$”，编译器会将该数字变量的值转换为十六进制的字符串，并将该十六进制的字符串代换“$”后的数字变量。

如果在逻辑变量前面有一个代换操作符“$”，编译器会将该逻辑变量代换为它的取值（真或假）。

如果在字符串变量前面有一个代换操作符“$”，编译器会将该字符串变量的值代换“$”后的字符串变量。

如果程序中需要字符“$”，则可以用“$$”来表示。汇编器将不进行变量替换，而是将“$$”作为“$”。

下面的两个例子说明了变量替换的过程。

```
; 直接的变量替换
       GBLS    add4ff
       ;
add4ff  SETS    "ADD  r4,r4,#0xFF"  ;给变量 add4ff 赋值
       $add4ff.00                   ;引用变量
       ; codes
       ADD  r4,r4,#0xFF00
   ; 有特殊符号的变量替换
          GBLS    s1
          GBLS    s2
          GBLS    fixup
          GBLA    count
          ;
count     SETA    14
s1        SETS    "a$$b$count"     ;s1 =a$b0000000E
s2        SETS    "abc"
fixup      SETS    "|xy$s2.z|"     ;fixup= |xyabcz|
|C$$code|   MOV     r4,#16         ;label= C$$code
```

4．程序标号（label）

在 ARM 汇编中，标号代表一个地址，段内标号的地址在汇编时确定，而段外标号地址值在链接时确定。根据标号的生成方式，程序标号分为以下三种。

- 程序相关标号（Program-relative labels）。
- 寄存器相关标号（Register-relative labels）[3]。
- 绝对地址（Absolute address）。

（1）程序相关标号

程序相关标号指位于目标指令前的标号或程序中的数据定义伪操作前的标号。这种标号在汇编时将被处理成 PC 值加上或减去一个数字常量。它常用于表示跳转指令的目标地址或代码段中所嵌入的少量数据。

（2）寄存器相关地址

这种标号在汇编时将被处理成寄存器的值加上或减去一个数字常量。它常被用于访问数据段中的数据。这种基于寄存器的标号通常用 MAP 和 FIELD 伪操作定义，也可以用 EQU 伪操作定义。

（3）绝对地址

[3] 在一些文献中将程序相关标号 Program－relative label 翻译为“基于 PC 的标号”，将寄存器相关标号 Register－relative label 翻译为基于寄存器的标号。

绝对地址是一个 32 位的数字量，使用它可以直接寻址整个内存空间。

5．局部标号

局部标号是一个 0～99 之间的十进制数字，可重复定义。局部标号后面可以紧接一个通常表示该局部变量作用范围的符号。局部变量的作用范围为当前段，也可以用伪操作 ROUT 来定义局部标号的作用范围。

局部标号在子程序或程序循环中常被用到，也可以配合宏定义伪操作（MACRO 和 MEND）来使程序结构更加合理。

在同一个段中，可以使用相同的数字命名不同的局部变量。默认情况下，汇编器会寻址最近的变量。也可以通过汇编器命令选项来改变搜索顺序。

局部变量命名语法如下。

```
n{routname}
```

局部变量引用的语法格式如下。

```
%{F|B}{A|T}n{routname}
```

其中，routname 为变量作用范围名称；%表示引用操作；F 指示汇编器只向前搜索；B 指示汇编器只向后搜索；A 指示汇编器搜索所有宏的嵌套。T 指示汇编器只搜索宏的当前层。

如果在引用过程中，没有指定 F 和 B，则汇编器先向后搜索，再向前搜索。

如果 A 和 T 没有指定，汇编器搜索所有从当前层次到宏最高层次，比当前层次低的层次不再搜索。

如果指定了 routname，汇编器向前搜索最近的 ROUT 操作，若 routname 与该 ROUT 伪操作定义的名称不匹配，汇编器报告错误并结束汇编。

10.3.3　汇编语言程序中的表达式和运算符

在汇编语言程序设计中经常使用各种表达式，表达式一般由变量、常量、运算符和括号构成。常用的表达式有数字表达式、逻辑表达式和字符串表达式。

下面分别介绍表达式中各元素。

1．字符串表达式

字符串表达式一般由字符串常量、字符串变量、运算符和括号构成。字符串由包含在双引号内的一系列字符组成。编译器所支持的字符串最大长度为 512 字节。

当在字符串中包含“$”或引号时，可以用“$$”表示“$”,用两个双引号表示一个双引号。

例如：

```
abc     SETS    "one "" double quote"
def     SETS    "one $$ dollar symbol"
```

上面的例子分别将字符串 abc 和 def 赋值为“one " double quote”和“one $ dollar symbol”。

字符串可以通过 SETA、SETL、SETS 伪操作对其赋值。

常用的与字符串表达式相关的运算符如下。

- LEN：计算字符串长度运算符。
- CHR：ASCII 码转换运算符。
- STR：字符串转换运算符。
- LEFT：字符串取左运算符。
- RIGHT：字符串取右运算符。
- CC：字符串连接运算符。

详见后面操作符一节。

下面的例子说明了如何使用字符串操作符给字符串变量赋值。

```
improb  SETS   "literal":CC:(strvar2:LEFT:4)
```

这个例子将字符串赋值为“literalatrv”。

2．整数表达式

整数表达式一般由数字常量、数字变量、数字运算符和括号构成。

整数表示式可以包含寄存器相关（register-relative）或程序相关（program-relative）表达式，这些表达式在编译时被汇编器翻译为地址无关数字常量。

整数表达式一般被计算为 32 位的整数，当此整数被定义为无符号数时，其取值范围为 0～2^{32}-1，当被定义为有符号数时，其取值范围为-2^{31}～2^{31}-1。汇编器认为-n 和 2^{32}-n 是相等的。对于关系操作，如比较两个数的大小，汇编器将其操作数看作无符号的数，也就是说“0>-1”对汇编器来说取值为“假（{FLASE}）”。

下面的例子说明了在程序中，如何对整数表达式进行操作。

```
a   SETA   256*256          ;将数字变量赋值为 256*256
    MOV    r1,#(a*22)       ;将数字表达式(a*22) 的值放入 r1
```

汇编语言中，整数数字量有以下几种形式。

- 十进制数（decimal-digis）
- “0x”+ 十六进制数（0xhexadecimal-digits）
- “&”+ 十六进制数（&hexadecimal-digits）
- n 进制数（n_base-n-digits）
- 字符（character）

其中，十进制数（decimal-digis）可以是“0”到“9”数字的任意组合；十六进制数（hexadecimal-digits）可以是“0”到“9”数字和字母“A”到“F”的任意组合；“n_”可以取 2 到 9，“base-n-digits”是在 n 进制下合法的任意数值；字符（character）可以是除单引号以外的所有字符。

下面的例子说明了整数表达式的基本用法。

```
a      SETA   34906
addr    DCD    0xA10E
```

```
        LDR     r4,=&1000000F
        DCD     2_11001010
c3      SETA    8_74007
        DCQ     0x0123456789abcdef
        LDR     r1,='A'         ;ARM 伪指令将整数 65（A 的 ASCII 码）存入寄存器
        ADD     r3,r2,#'\''     ;将整数 39（字符“/”的 ASCII 码）加到 r2，结果存入 r3
```

3．浮点数字量表达式

浮点数字量有以下几种形式。

- {-}digitsE{-}digits。
- {-}{digits}.digits{E{-}digits}。
- 0xhexdigits。
- &hexdigits。

其中，digits 为十进制数，要在其后加上字母 E（大写或小写）来表示其指数；hexdigits 为十六进制数。

单精度浮点数的表示范围为 1.17549435e−38～3.40282347e+38；双精度浮点数的表示范围为 2.22507385850720138e-308～1.79769313486231571e+308。

下面的例子说明了浮点数据量的基本用法。

```
    DCFD    1E308,-4E-100
    DCFS    1.0
    DCFD    3.725e15
    LDFS    0x7FC00000              ;
    LDFD    &FFF0000000000000       ;
```

4．逻辑表达式

逻辑表达式一般由逻辑量、逻辑运算符和括号构成，其表达式的运算结果为真或假。与逻辑表达式相关的运算符有“=”、“>”、“<”、“>=”、“<= ”、“/=”、“ <>” 运算符和“LAND”、“LOR”、“LNOT”及“LEOR”运算符。

5．程序或寄存器相关表达式

寄存器相关表达式的值等于指定寄存器的值加上或减去一个数字表达式。

程序相关表达式的值等于程序计数器 PC 的值加上或减去一个数字表达式的值。此种表达式通常由程序中的标号与一个数字表达式组成。

下面的例子说明了程序或寄存器相关表达式的基本使用方法。

```
        LDR     r4,=data+4*n        ;n 是汇编时取值变量
        ; code
        MOV     pc,lr
data    DCD     value0
```

```
        ; n-1 个 DCD 伪操作
        DCD     valuen              ;data+4*n 指向此
        ;更多 DCD 伪操作
```

6．汇编中的操作符

（1）操作符的优先级

在汇编语言程序设计中，表达式包含一个扩展的操作符集，这些操作符和高级语言中的运算符十分接近。其运算次序遵循如下的优先级。

① 优先级相同的双目运算符的运算顺序为从左到右。

② 相邻的单目运算符的运算顺序为从右到左，单目运算符的优先级高于其他运算符。

③ 括号运算符的优先级最高。

汇编语法的操作符优先级和 C 语言中的不完全相同。例如在汇编中，下面的汇编语言（1+2：SHR：3）相当于（1+（2：SHR：3）），而在 C 语言中，运算则变为（（1+2）>>3）=0。类似于这样的操作，在使用时要特别注意。

注意 为了保证表达式运算结果的正确，建议使用“()”来避免异义。

表 10.4 列出了汇编操作符的优先级以及对应的 C 语言运算符。

表 10.4 汇编操作符优先级

汇编操作符	C 语言运算符
单目运算	单目运算
* / :MOD:	* / %
字符串操作	n/a
:SHL::SHR::ROR::ROL:	<< >>
+ - :AND: :OR: :EOR:	+ — $\|
= > >= < <= /= <>	== > >= < <= !=
:LAND: :LOR: :LEOR:	&& ‖

说明 表 10.3 是按操作符的优先级从上到下排列的。

C 语言运算符优先级从高到低排列如下。

- 单目运算
- * / %
- + - (as binary operators)
- << >>
- < <= > >=
- == !=
- &
- ^

- |
- &&
- ||

（2）单目运算

最高优先级的单目运算在表达式中最先被计算。单目操作符写在操作数的前面。运算顺序为从右到左。

表 10.5 列出了汇编中单目运算操作符及其返回值。

表 10.5　　汇编中单目运算操作符及其返回值。

操　作　符	使　　用	描　　述
:CHR:	:CHR:A	返回字母 A 的 ASCII 码
:LOWERCASE	:LOWERCASE:string	将给定字符串中的所有大写字母变成小写
REVERSE_CC	:REVERSE_CC:cond_code	对条件码取反
:STR:	:STR:A	将一个数字量或逻辑表达式转换成串
:UPPERCASE:	:UPPERCASE:string	将给定字符串中的所有小写字母变成大写
?	？A	返回定义符号 A 的代码行所生产代码行的字节数

续表

操　作　符	使　　用	描　　述
+和–	+A 和-A	单目加和单目减，操作数为数学或程序相关表达式
:BASE:	:BASE:A	如果 A 是程序或寄存器相关表达式，:BASE:返回基址寄存器的编号
:CC_ENCODING:	:CC_ENCODING:cond_code	返回条件码中的数字值
:DEF:	:DEF:A	判断 A 是否被定义，如果被定义返回{TRUE}；如果没有定义返回{FALSE}
:INDEX:	:INDEX:A	如果 A 是寄存器相关表达式，:INDEX:返回 A 相对于寄存器的偏移量，常用在宏操作中
:LEN:	:LEN:A	字符串 A 的长
:LNOT:	:LNOT:A	逻辑表达式 A 的值取反
:NOT:	:NOT:A ~A	A 的值按位取反
:RCONT:	:RCONT:Rn	返回寄存器编号，0～15 对应寄存器 r0～r15

（3）双目运算

ARM 汇编中将双目运算符放在两个操作数中间。一般情况下，双目运算的优先级低于单目运算。下面将以操作符的优先级为序分别介绍各操作符。

注意　操作符的优先级与 C 语言中操作符优先级顺序略有不同，详见单目运算一节。

表 10.6 列出了乘法相关操作符。

表 10.6 乘法相关操作符

操 作 符	别 名	使 用	说 明
*		A*B	乘法操作
/		A/B	除法操作
:MOD:	%	A:MOD:B	以 B 为除数对 A 取模

乘法相关操作符包括乘、除、取模运算，在双目运算中具有最高优先级。这些运算的操作数只能是数字表达式。

表 10.7 列出了字符串相关操作符。

表 10.7 字符串操作符

操 作 符	使 用	说 明
:CC:	A:CC:B	连接两个字符串
:LEFT:	A:LEFT:B	返回字符串 A 最左端 B 长度的字符，操作数 A 必须为字符串，B 必须为整数表达式
:RIGHT:	A:RIGHT:B	返回字符串 A 最右端 B 长度的字符，操作数 A 必须为字符串，B 必须为整数表达式

表 10.8 列出了移位操作符。移位操作中两个操作数均为数字表达式。

表 10.8 移位操作符

操 作 符	别 名	使 用	说 明
:ROL:		A:ROL:B	A 循环左移 B 位
:ROR:		A:ROR:B	A 循环右移 B 位
:SHL:	<<	A:SHL:B	A 左移 B 位
:SHR:	>>	A:SHR:B	A 右移 B 位

注意 SHR 是逻辑右移，不影响符号位。

表 10.9 列出了所有加、减、逻辑操作符。

表 10.9 加减运算操作符

操 作 符	别 名	使 用	说 明
+		A+B	A 加上 B
−		A−B	从 B 中减去 A
:AND:	&&	A:AND:B	A 和 B 按位与

:EOR:	^	A:EOR:B	A 和 B 按位异或
:OR:	\|	A:OR:B	A 和 B 按位或

加、减运算的操作数均为数字表达式。逻辑运算的表达式为数字表达式，此运算按位操作产生结果。

表 10.10 列出了 ARM 汇编中的关系符。关系操作符用于表示两个同类表达式之间的关系。关系符的两个操作数必须为同种类型的操作数。操作数可以是数字变量、程序相关表达式、寄存器相关表达式或字符串。

表 10.10　　关系操作符

操　作　符	别　　名	使　　用	说　　明
=	= =	A=B	判断 A 是否等于 B
>		A>B	判断 A 是否大于 B
>=		A>=B	判断 A 是否大于等于 B
<		A<B	判断 A 是否小于 B
<=		A<=B	判断 A 是否小于等于 B
/=	<> !=	A/=B	判断 A 是否不等于 B

表 10.11 列出了汇编语言中的逻辑操作符。逻辑操作符进行两个逻辑表达式之间的基本逻辑操作。操作的结果为{FALSE}或{TURE}。

表 10.11　　逻辑操作符

操　作　符	使　　用	说　　明
:LAND:	A:LAND:B	A 和 B 做逻辑与

续表

操　作　符	使　　用	说　　明
:LEOR:	A:LEOR:B	A 和 B 做逻辑异或
:LOR:	A:LOR:B	A 和 B 做逻辑或

10.3.4　汇编语言预定义寄存器和协处理器

ARM 汇编器对 ARM 的寄存器和协处理器进行了预定义（包括 APCS 对 r0～r15 寄存器的定义），所有的寄存器和协处理器名都是大小写敏感的。

1．预定义寄存器名

下面列出了被 ARM 汇编器预定义的寄存器名。

- r0～r15 和 R0~R15（15 个通用寄存器）。
- a1~a4（参数、结果或临时寄存器，同 r0～r3）。

- v1～v8（变量寄存器，同 r4～r11）。
- sb 和 SB（静态基址寄存器，同 r9）。
- sl 和 SL（栈顶指针寄存器，同 r10）。
- fp 和 FP（帧指针寄存器，同 r11）。
- ip 和 IP（过程调用中间临时寄存器，同 r12）。
- sp 和 SP（栈指针寄存器，同 r13）。
- lr 和 LR（连接寄存器，同 r14）。
- pc 和 PC（程序计数器，同 r15）。

2．预定义程序状态寄存器名

下面列出了 ARM 汇编器预定义的程序状态寄存器的名称。

- cpsr 和 CPSR（当前程序状态寄存器）。
- spsr 和 SPSR（保留程序状态寄存器）。

3．预定义的浮点寄存器名

下面列出了 ARM 汇编器预定义的浮点运算寄存器。

- s0～s31 和 S0~S31（VFP 单精度浮点运算寄存器）。
- d0～d15 和 D0~D15（VFP 双精度浮点运算寄存器）。

注意 FPA 的寄存器 f0～f7 和 F0～F7 已不再使用。

4．预定义的协处理器名

下面列出了 ARM 汇编器预定义的协处理器名和协处理器寄存器名。

- p0～p15（预定义的协处理器 0～15 的名称）。
- c0～c15（预定义的协处理器寄存器 0～15 的名称）。

10.3.5 汇编语言内置变量

ARM 汇编器中定义了一些内置变量，这些内置变量不能使用伪指令设置（如，SETA、SETL、SETS 等），一般用于程序的条件汇编控制。

下面的例子显示了如何使用内置变量控制程序的执行流程。

```
If  {CONFIG}=16            ;若为 Thumb 代码则执行 If 后的语句
;codes
else
;codes
endif
b                          ;程序结束
```

下面介绍由 ARM 汇编器预定义的内置变量。

- {ARCHITECTURE}：选定的 ARM 体系结构的值，如 3，3M，4，4T。
- {AREANAME}：当前段名。
- {ARMASM_VERSION}：ARM 编译器 ARMASM 的变量号。
- |ads$version|：ARM 编译器 ARMASM 的变量号，同{ARMASM_VERSION}。
- {CODESIZE}：如果当前指令为 ARM 指令，该内置变量取值为 32，如果当前指令为 Thumb 指令，该内置变量取值为 16，同{CONFIG}。
- {COMMANDLINE}：当前命令行内容。
- {CONFIG}：如果当前指令为 ARM 指令，该内置变量取值为 32，如果当前指令为 Thumb 指令，该内置变量取值为 16，同{CODESIZE}。
- {CPU}：所使用的 CPU 名称。默认为 ARM7TDMI。如果在编译命令行中使用“-CPU”选项确定 CPU 类型，则该值为“Generic ARM”。
- {ENDIAN}：如果编译器在大端模式下，其值为“big”；如果在小端模式下，其值为“little”。
- {FPIC}：默认为{FALSE}，如果设置了“/fpic”选项，其值为{TRUE}。
- {FPU}：所选 fpu 协处理器的名字。默认为“softVFP”。
- {INPUTFILE}：当前源文件名。
- {INTER}：默认为{FALSE}，如果设置了“/inter”选项，其值为{TRUE}。
- {LINENUM}：目前源文件行号。
- {NOSWST}：默认为{FALSE}，如果设置了“/noswst”选项，其值为{TRUE}。
- {OPT}：保存当前设置的列表选项。伪操作 OPT 用来保存当前列表选项，改变选项值，或恢复原始值。
- {PC} 或“.”：当前程序地址值。
- {PCSTOREOFFSET}：指令 STR pc,[...]和 STM Rb,{..., pc}与存储的 PC 值之间的偏移量。
- {ROPI}：默认为{FALSE}，如果设置了“/ropi”选项，其值为{TRUE}。
- {RWPI}：默认为{FALSE}，如果设置了“/rwpi”选项，其值为{TRUE}。
- {SWST}：默认为{FALSE}，如果设置了“/swst”选项，其值为{TRUE}。
- {VAR}或@：存储区位置寄存器的当前值。

10.3.6 汇编语言的程序结构

在 ARM（Thumb）汇编语言程序中以程序段为单位组织代码。段是相对独立的指令或数据序列，具有特定的名称。段可以分为代码段（Code Section）和数据段（Data Section），代码段的内容为执行代码，数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段，当程序较长时，可以分割为多个代码段和数据段，多个段在程序编译链接时最终形成一个可执行的映像文件。

可执行映像文件通常由以下几部分构成。

- 一个或多个代码段，代码段的属性为只读。
- 零个或多个数据段，数据段的属性为可读写。数据段可是被初始化的数据段或没有被初始化的数据段（ZI，zero initialized）。

链接器根据系统默认或用户设定的规则，将各个段安排在存储器中的相应位置。因此源

程序中段之间的相对位置与可执行的映像文件中段的相对位置一般不会相同。

以下是一个汇编语言源程序的基本结构。

```
        AREA   Init，CODE，READONLY
        ENTRY
Start
        LDR        R0，=0x3FF5000
        LDR        R1，0xFF
        STR        R1，[R0]
        LDR        R0，=0x3FF5008
        LDR        R1，0x01
        STR        R1，[R0]
        ……
        END
```

在汇编语言程序中，用 AREA 伪操作定义一个段，并说明所定义段的相关属性，本例定义一个名为 Init 的代码段，属性为只读。ENTRY 伪操作标识程序的入口点，接下来为指令序列，程序的末尾为 END 伪指令，该伪操作告诉编译器源文件的结束，每一个汇编程序段都必须有一条 END 伪操作，指示代码段的结束。

10.3.7 汇编语言子程序调用

在 ARM 汇编语言程序中，子程序的调用一般是通过 BL 指令来实现的。在程序中，使用指令“BL 子程序”名即可完成子程序的调用。

该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器 LR 中，同时将程序计数器 PC 指向子程序的入口点。当子程序执行完毕需要返回调用处时，只需要将存放在 LR 中的返回地址重新拷贝给程序计数器 PC 即可。在调用子程序的同时，也可以完成参数的传递和从子程序返回运算的结果，通常可以使用寄存器 R0～R3 完成。

注意 不同编译器编译的代码间的相互调用，要遵循 AAPCS（ARM Architecture）。详见 ARM 编译工具手册。

以下是使用 BL 指令调用子程序的汇编语言源程序的基本结构：

```
AREA        Init，CODE，READONLY
ENTRY
Start
LDR     R0，=0x3FF5000
LDR     R1，0xFF
STR     R1，[R0]
LDR     R0，=0x3FF5008
LDR     R1，0x01
STR     R1，[R0]
BL      PRINT_TEXT
```

```
……
PRINT_TEXT
……
      MOV       PC，BL
……
END
```

10.4 ARM 汇编编译器的使用

armasm 是 ARM 汇编语言的交叉编译器，本节将详细介绍它的使用方法。

armasm 命令行语法格式如下。

```
armasm options inputfile
```

在 armasm 命令中，除了文件名区分大小写之外，其他的参数都不区分大小写。option 可是汇编器规定选项中的一个或多个的组合，多个选项用空格分开。下面详细介绍 armasm 的各参数。

- --16：告诉汇编器当前程序是 Thumb 指令程序，使用旧的 Thumb 语法，与在源程序开头使用伪操作 CODE16 意义相同。使用--thumb 选项指定当前程序是使用 ARM 语法的 Thumb 或 Thumb-2 源程序。
- --32：告诉汇编器所处理的源程序是 ARM 指令的程序。此选项为汇编器的默认选项。
- --apcs [qualifiers]：该选项告诉汇编器是否使用 AAPCS 标准编译源程序。详见 AAPCS 一节。
- --arm：同--32 选项。
- --bigend：告诉汇编器将源程序按大端模式编译。汇编器默认为小端模式（littleend）。
- --brief_diagnostics：控制输出诊断信息。详见控制诊断信息输出一节。
- --littleend：告诉编译器将源程序按小端模式编译。这是汇编器的默认选项。
- --checkreglist：告诉汇编器检测 RLIST、LDM、STM 指令的寄存器列表是否按升序排列。使用此选项后，如果源程序中寄存器没有按升序排列，汇编器将给出警告信息。
- --cpu name：告诉汇编器允许程序所使用的 CPU 型号，详见 CPU 名一节。
- --debug：告诉编译器编译时产生 DWARF 格式的调试信息表。
- --depend dependfile：告诉编译器将程序的依赖关系列表输出到指定的 dependfile 文件中。当使用 makefile 文件对程序进行编译时，该选项十分有用。
- --diag_[error | remark | warning | suppress | style]：详见控制诊断信息输出一节。
- --dllexport_all：告诉编译器产生全局变量的动态可见列表（dynamic visibility），在将源程序编译成 DLL 文件时使用该选项。
- --dwarf2：和--debug 选项配合使用。告诉编译器编译时产生 DWARF2 格式的调试信息表，当使用了--debug 选项时，该选项为默认选项。
- --dwarf3：和--debug 选项配合使用。告诉编译器编译时产生 DWARF3 格式的调试

信息表。

- -m：告诉编译器将源文件的依赖关系列表输出到标准输出设备上。
- --md：告诉编译器将源文件的依赖关系列表输出到 inputfile.d 文件中。
- --errors errorfile：告诉编译器将编译的错误信息输出到 errorfile 文件中。
- --exceptions：详见指示编译器产生异常向量表一节。
- --exceptions_unwind：详见指示编译器产生 anwind 异常向量表一节。
- --fpmode model：详见 10.4.3。
- --fpu name：选择指定目标系统中浮点运算单元的体系结构。
- -i dir [,dir]…：为源文件搜索增加路径，如果要搜索的源文件路径已被此选项指定，那么在使用伪操作 GET、INCLUDE、INCBIN 包含源文件时，将不必指定搜索路径。
- --keep：指定汇编器将局部符号保留在目标文件的符号表中，供调试器进行调试时使用。
- --list [listingfile]：告诉汇编器将汇编过程中产生的汇编程序列表保存到列表文件 listingfile 文件中。
- --maxcache *n*：指定最大的源程序 cache[4]大小，默认为 8MB。
- --memaccess attributes：确定目标系统的内存属性。详见内存访问属性一节。
- --no_cache：禁止源程序 cache。默认情况下源程序 cache 是打开的。
- --no_esc：禁止 C 风格的特殊符号，如“\n”、“\t”。
- --no_exceptions：详见异常向量表产生一节。
- --no_exceptions_unwind：详见异常向量表产生一节。
- --no_hide_all：控制符号的可见性。
- --no_regs：告诉编译器不使用汇编器预定义的寄存器名。
- --no_warn：禁止警告信息。
- -o filename：给输出的目标文件命名。
- --predefine“directive”：告诉编译器预执行 SET 伪操作。详见预执行 SET 伪操作一节。
- --split_ldm：详见 LDM 和 STM 指令。
- --thumb：告诉编译器将源文件编译为符合 ARM 语法的 Thumb 指令。此编译选项和在源文件头使用 THUMB 伪操作效果相同。
- --unsafe：降低汇编器的警告级别。详见控制针对信息输出一节。
- --via file：指示汇编器从指定文件 file 中读取各选项信息。
- inputfile：为输入的源程序，必须是 ARM 或 Thumb 汇编源程序。

下面各节详细介绍以上选项中常用到的选项。

10.4.1 选项说明列表

在命令行输入下面的汇编命令，可使汇编器输出所有可用选项类别。

[4] 源程序 Cache 是值 armasm 在第一遍扫描时将源程序缓存到内存中，在第二遍扫描时，从内存中读取该源程序。

```
armasm -help
```

10.4.2 过程调用标准 AAPCS

为了使不同编译器编译的程序之间能够相互调用，必须为子程序间的调用规定一定的规则。AAPCS 就是这样一个标准。所谓 AAPCS，其英文全称为 Procedure Call Standard for the ARM Architecture (AAPCS)，即 ARM 体系结构过程调用标准。它是 ABI（Application Binary Interface (ABI) for the ARM Architecture (base standard) [BSABI]）标准的一部分。

可以使用“--apcs”选项告诉编译器将源代码编译成符号 AAPCS 调用标准的目标代码。

注意 使用“--apcs”选项并不影响代码的产生，编译器只是在各段中放置相应的属性，标识用户选定的 AAPCS 属性。

与 AAPCS 相关的编译/汇编选项有以下几种。

- none：指定输入文件不使用 AAPCS 规则。
- /interwork：指定输入文件符合 ARM/Thumb 交互标准。
- /nointerwork：指定输入文件不能使用 ARM/Thumb 交互。这是编译器默认选项。
- /ropi：指定输入文件是位置无关只读文件。
- /noropi：指定输入文件是非位置无关只读文件。这是编译器默认选项。
- /pic：同/ropi。
- /nopic：同/noropi。
- /rwpi：指定输入文件是位置无关可读可写文件。
- /norwpi：指定输入文件是非位置无关可读可写文件。
- /pid：同/rwpi。
- /nopid：同/norwpi。
- /fpic：指定输入文件编译成位置无关只读代码。代码中地址是 FPIC 地址。
- /swstackcheck：编译过程中对输入文件使用堆栈检测。
- /noswstackcheck：编译过程中对输入文件不使用堆栈检测。这是编译器默认选项。
- /swstna：如果汇编程序对于是否进行数据栈检查无所谓，而与该汇编程序连接的其他程序指定了选项/swst 或选项/noswst，这时该汇编程序使用选项/swstna。

10.4.3 浮点模式选项

“--fpmode model”选项指定所选的浮点模式。可使用的浮点模式有以下几种。

- ieee_full：所有的浮点操作符号 IEEE 标准，其中包括单精度浮点操作和双精度浮点操作。浮点模式可以在使用时动态选择。使用该选项，编译器将使用下面的预定义符号。

__FP_IEEE；

__FP_FENV_EXCEPTIONS；

__FP_FENV_ROUNDING；

__FP_INEXACT_EXCEPTION。

- ieee_fixed：符合 IEEE 标准的浮点运算异常处理规则。使用该选项，编译器将使用下

面的预定义符号。

__FP_IEEE；

__FP_FENV_EXCEPTIONS。

- ieee_no_fenv：和 JAVA 兼容的浮点运算算法选择。预定义的符号为__FP_IEEE。
- std：和 C 和 C++兼容的浮点运算算法选择。这是编译器默认选项。
- fast：快速浮点运算选项。使用该选项将影响浮点运算的精度。

10.4.4 为 CPU 命名选项

使用“--cpu name”选项为目标程序使用的 CPU 命名。其中 name 的取值为 4T、5TE 或 6T2。编译器的默认值为 ARM7TDMI。

下面的选项使编译器列出所有当前可使用的 CPU 名。

```
armasm --cpu list
```

10.4.5 为 FPU 命名选项

选项“--FPU”指定所使用的浮点运算单元的结构。

10.5 ARM 汇编程序设计举例

在本节中通过一些例子来说明 ARM 中伪操作及指令的基本用法。

10.5.1 条件跳转及循环

1．ALU 状态标志

所有 ARM 指令都可以条件执行。大部分 ARM 指令集和 Thumb-2 指令集的数据处理指令都可以选择是否根据指令的执行结果设置 ALU 的状态标志位。



注意 较早的 ARM 体系结构中使用的 Thumb 指令不能选择是否更新 ALU 的标志位。当数据处理指令执行完后，处理器自动根据指令的执行结果更新状态标志。

较早的 Thumb-2 指令只有跳转指令可以条件执行。新的体系结构中的 Thumb-2 指令可以 IT（if-then）标识使程序条件执行。

更详细的介绍请参加本书的指令集部分。

2．ARM 状态下的条件执行

在程序状态寄存器 CPSR 中保存着以下 4 个 ALU 状态标志。

- N：当指令的执行结果为负时，该位置 1。
- Z：当指令的执行结果为零时，该位置 1。
- C：当指令的执行结果有进位时，该位置 1。

- V：当指令的执行结果溢出时，该位置 1。

当加法操作的结果大于等于 2^{32} 或加法操作的结果为负时，进位标志 C 置位。

当加法、减法、比较操作结果大于等于 2^{31} 或小于-2^{32} 时，溢出标志 V 置位。

在 ARM 指令后增加条件域可以使指令条件执行，各条件码的含义和助记符如表 10.11 所示。可条件执行的指令可以在其助记符的扩展域加上条件码助记符，从而在特定条件下执行。

表 10.11　　指令的条件码

条　件　码	助记符后缀	标　　志	含　　义
0000	EQ	Z 置位	相等
0001	NE	Z 清零	不相等
0010	CS	C 置位	无符号数大于或等于
0011	CC	C 清零	无符号数小于
0100	MI	N 置位	负数
0101	PL	N 清零	正数或零
0110	VS	V 置位	溢出
0111	VC	V 清零	未溢出
1000	HI	C 置位 Z 清零	无符号数大于
1001	LS	C 清零 Z 置位	无符号数小于或等于
1010	GE	N 等于 V	带符号数大于或等于
1011	LT	N 不等于 V	带符号数小于
1100	GT	Z 清零且（N 等于 V）	带符号数大于
1101	LE	Z 置位或（N 不等于 V）	带符号数小于或等于
1110	AL	忽略	无条件执行

默认情况下，ARM 指令并不会更新 ARM 寄存器 cpsr 中的 N、Z、C、V 标志。对大多数指令，若要更新这些标志需要对指令助记符加后缀 S。但 CMP 指令不需要 S 后缀就更新这些标志位。

下面的一段程序，说明了指令的 S 后缀和条件执行的过程。

```
ADD     r0, r1, r2      ;r0 = r1 + r2，不更新标志位
ADDS    r0, r1, r2      ;r0 = r1 + r2，并且更新标志位
ADDSCS  r0, r1, r2      ;如果进位标志位 C=1，r0 = r1 + r2，并且更新标志位
CMP     r0, r1          ;根据 r0-r1 的结果更新标志位
```

从上面的例子可以看出，无论更新标志位标志“S”是否设置，CMP 指令自动更新标志位。

3．条件执行的例子

通过组合使用条件执行和条件标志设置，可是简单地实现分支语句，不需要任何分支指令。这样可以改善性能，因为分支指令会占用较多的周期数；同时这样做也可以减小代码尺寸，提高代码密度。

下面是一段 C 语言程序，该程序实现了著名的 Euclid 最大公约数算法。

```
int gcd(int a, int b)
{
     while (a != b)
       {
         if (a > b)
                a = a - b;
         else
                b = b - a;
       }
     return a;
}
```

用 ARM 汇编语言重写来重写这个例子，如下所示。

【程序 1】

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        SUB     r0, r0, r1
        B       gcd
less
        SUB     r1, r1, r0
        B       gcd
End
```

充分地利用条件执行修改上面的例子，得到【程序 2】。

【程序 2】

```
gcd
       CMP     r0, r1
       SUBGT   r0, r0, r1
       SUBLT   r1, r1, r0
       BNE     gcd
```

【程序 1】仅使用了分支指令，【程序 2】充分利用了 ARM 指令条件执行的特点，仅使用

了 4 条指令就完成了全部算法。这对提供程序的代码密度和执行速度十分有帮助。

事实上，分支指令十分影响处理器的速度。每次执行分支指令，处理器都会排空流水线，重新装载指令。

注意 新的 ARM 处理器如 ARM10 和 StrongARM 都有分支预测硬件。只有当分支预测硬件失败时，处理器才排空流水线，重新装载指令。

表 10.12 和 10.13 分别总结了【程序 1】和【程序 2】在 ARM7 上的执行过程（假设 r0=1、r1=2）。

表 10.12 程序 1 执行过程

r0：a	R1：b	指　　令	执行周期（ARM7）
1	2	CMP r0, r1	1
1	2	BEQ end	1 (未执行)
1	2	BLT less	3
1	2	SUB r1, r1, r0	1
1	2	B gcd	3
1	1	CMP r0, r1	1
1	1	BEQ end	3
			Total = 13

表 10.13 程序 2 执行过程

r0：a	R1：b	指　　令	执行周期（ARM7）
1	2	CMP r0, r1	1
1	2	SUBGT r0,r0,r1	1 (未执行)
1	1	SUBLT r1,r1,r0	1
1	1	BNE gcd	3
1	1	CMP r0,r1	1
1	1	SUBGT r0,r0,r1	1 (未执行)
1	1	SUBLT r1,r1,r0	1 (未执行)
1	1	BNE gcd	1 (未执行)
			Total = 10

从上面的例子可以看出，利用条件执行执行可以实现大部分条件语句，比使用条件分支指令效率高很多。

10.5.2 传送指令程序设计

1. 加载立即数

通常，使用 MOV 和 MVN 指令向寄存器加载常量，但由于单条 MOV 或 MVN 指令只能包含 8 位立即数，向寄存器加载立即数需要一些特殊的操作。

注意 在 ARMv6T2 体系结构及以上版本中，可以使用 MOV32 伪操作将一个 32 位立即数加载到寄存器。

下面分别介绍加载立即数的不同方法。

（1）使用 MOV 和 MVN 指令

在 ARM 或 Thumb-2 状态下，可以使用 MOV 和 MVN 指令将符合一定规则的常数加载到寄存器。在 16 位的 Thumb 指令下，可以将 0～255 的任意常数加载到寄存器。表 10.14 列出了在 ARM 状态下可直接加载到寄存器的立即数。

表 10.14 ARM 状态合法立即数

二 进 制	十 进 制	步骤	十六进制	MVN 指令值	注释
00000000000000000000000abcdefgh	0～255	1	0～0xFF	-1～-256	
00000000000000000000abcdefgh00	0～1020	4	0～0x3FC	-4～-1024	
0000000000000000000abcdefgh0000	0～4080	16	0～0xFF0	-16～-4096	
000000000000000000abcdefgh000000	0～16320	64	0～0x3FC0	-64～-16384	
	...	...	...	...	
abcdefgh000000000000000000000000	0～255×2^{24}	2^{24}	0～0xFF000000	1～256×“-2^{24}”	
cdefgh000000000000000000000000ab	(bit pattern)	—	—	(bit pattern)	②
efgh000000000000000000000000abcd	(bit pattern)	—	—	(bit pattern)	②
gh000000000000000000000000abcdef	(bit pattern)	—	—	(bit pattern)	②
00000000000000000000abcdefghijkl	0～4095	1	0～0xFFF	—	③

注释

① 在 ARM 数据操作指令中，除 MVN 外，其他指令不能直接操作 MVN 的值，即 MVN 列所列出的立即数在其他数据操作指令中可能为非法操作数。

② 表中所列数据为 ARM 状态合法数据，对 Thumb－2 状态不一定适用。

③ 这些值只能在 ARMv6T2 体系结构及其以上版本中适用。

ARM 状态下立即数的使用要符合以下规则。

① 每个立即数由一个 8 位的常数循环右移偶数位得到。

注意 这样得到的立即数对任何数据处理指令都是合法的。

② MVN 指令向寄存器加载一个合法立即数的“按位反”，即-（n+1）。其中 n 为 MOV 指令可以加载的合法立即数。

③ ARMv6T2 体系结构及其以上版本，可以加载 12 位的立即数，但 MVN 指令不能加载此 12 位立即数的“按位反”。

在 ARMv6T2 体系结构及其以上版本的 Thumb 状态下，可以加载的合法立即数有以下规则。

① 32 位的 MOV 指令可以加载的立即数符合以下规则。

- 任意 8 位立即数（范围 0x0～0xff）。
- 任意 8 位立即数向左移任意位。
- 任意 8 位立即数重复 4 次填充 32 位寄存器。
- 使用任意 8 位立即数填充一个 32 位寄存器的第 0 和第 2 字节，其余两个字节填充 0。
- 使用任意 8 位立即数填充一个 32 位寄存器的第 1 和第 3 字节，其余两个字节填充 0。

注意 通过上述方法得到的立即数，在其他数据操作指令中同样合法。

② MVN 指令向寄存器加载一个合法立即数的“按位反”，即-（n+1）。其中 n 为 MOV 指令可以的加载的合法立即数。

③ 可以加载任意 12 位的立即数，但 MVN 指令不能加载此 12 位立即数的“按位反”。

表 10.15 列出了在 Thumb-2 状态下可直接加载到寄存器的立即数。

表 10.15 Thumb 状态合法立即数

二 进 制	十进制	步骤	十 六 进 制	MVN 指令值	注释
000000000000000000000000abcdefgh	0～255	1	0～0xFF	-1 to -256	
000000000000000000000000abcdefgh0	0～510	2	0～0x1FE	-2 to -512	
000000000000000000000000abcdefgh00	0～1020	4	0～0x3FC	-4 to -1024	
	...	...	...	...	
0abcdefgh0000000000000000000000			0～0x7F800000		
abcdefgh0000000000000000000000			0～0xFF000000		
abcdefghabcdefghabcdefghabcdefgh	(bit pattern)	-	0xXYXYXYXY	-	
00000000abcdefgh00000000abcdefgh	(bit pattern)	-	0x00XY00XY	0xFFXYFFXY	
abcdefgh00000000abcdefgh00000000	(bit pattern)	-	0xXY00XY00	0xXYFFXYFF	
000000000000000000abcdefghijkl	0～4095	1	0～0xFFF	-	①

（2）使用 MOV32 伪操作

ARMv6T2 体系结构中，ARM 和 Thumb-2 指令集包括下面两条数据传送指令。

- MOV 指令：可以加载任意 8 位立即数到 32 位寄存器。
- MOVT 指令：可以加载任意 16 位立即数（0x0～0xffff）到 32 寄存器的高 16 位或低 16 位，而不改变余下的位。

可以使用组合的 MOV 和 MOVT 指令加载任意 32 位立即数到寄存器，也可以使用伪操作 MOV32 实现上述两条指令的组合功能。汇编器在扫描源代码时，自动将伪操作 MOV32 编译成 MOV 加 MOVT 指令的组合。

（3）使用 LDR 伪操作

LDR Rd,=const 伪操作可以将任意 32 位立即数加载到寄存器。可以使用此伪操作将超出 MOV 和 MVN 指令操作范围的立即数加载到寄存器。LDR 的效率很高，如果立即数可以由指令中的<shifter_operand>表示，则汇编器生成相应的 MOV 或 MVN 指令；如果立即数不能由<shifter_operand>直接表示，则汇编器将该立即数放到一个缓冲池（literal pool），并生成一条将该缓冲池内容加载到目标寄存器的 LDR 指令。例如，

```
    LDR     rn, [pc, #offset to literal pool]
```

该指令从地址[pc, #offset to literal pool]向 Rn 寄存器加载一个字。因此，必须确保在该 LDR 指令的访问范围内，存在一个可用的缓冲池。汇编器会在每个段后面添加一个缓冲池。对于 ARM 指令集，计数器 PC 的偏移量必须小于 4KB，对于 Thumb 指令，PC 的偏移量必须小于 1KB。通常，汇编器会在 LDR 伪操作后寻找可用的缓冲池，但是，如果 LDR 指令与默认缓冲池距离太远，则汇编器将会报错。此时必须在 LDR 指令上下 4KB（Thumb 为 1KB）之间用 LTORG 伪操作显式地在代码段中添加缓冲池，而且由于缓冲池在代码段中，必须确保它不会被处理器作为指令而加以执行。通常将其紧跟在无条件跳转指令后面。

下面是使用 LDR 伪操作加载立即数的例子。

```
            AREA    Loadcon, CODE, READONLY
            ENTRY                   ;函数入口
    start   BL      func1           ;跳转到 func1
            BL      func2           ;跳转到 func2
    stop    MOV     r0, #0x18       ;angel_SWIreason_ReportException 为 SWI 调用准备参
数
            LDR     r1, =0x20026    ;调用 SWI 的 ADP_Stopped_ApplicationExit 功能
            SWI     0x123456        ;调用 ARM semihosting SWI
    func1
            LDR     r0, =42         ;将立即数 42 存入
            LDR     r1, =0x55555555
                    ;将立即数 0x55555555 存入 r1,该指令被编译为 LDR R1,[PC,#offset]
            LDR     r2, =0xFFFFFFFF         ;将 0xFFFFFFFF 存入 r2
            BX      lr
            LTORG                           ;缓存池 1 contains
                                            ;该缓存池中包含立即数 0x55555555
    func2
            LDR     r3, =0x55555555
                    ;将立即数存入 r3，该伪操作被编译为 LDR R3,[PC,#offset]
            ; LDR r4, =0x66666666           ;将立即数 0x66666666 存入 r4
            BX      lr
    LargeTable
```

```
        SPACE   4200                ;申请 4200 字节的内存空间并初始化为零
        END                         ;Literal Pool 2 is empty
```

（4）加载浮点常数

ARM 体系结构中，允许加载单精度和双精度的浮点数。详细信息请参见 FLD 伪操作。

2．加载程序地址

通常在编写程序时需要加载程序地址。这些地址包括变量地址、字符串地址或程序跳转表的入口地址。这些地址通常是基于程序计数器 PC 或某个基址寄存器的偏移量。

下面分别介绍加载程序地址的不同方法。

（1）使用 ADR 和 ADRL 伪操作

使用 ADR 和 ADRL 伪操作可以直接向寄存器加载程序地址。ADR 和 ADRL 伪操作的操作数可以是程序相关表达式，汇编器在编译时，将此表达式转换成相对 PC 或寄存器的偏移量加载到目标寄存器。

注意 ADR 和 ADRL 伪操作所加载的地址是有一定限制的。加载的标号地址必须和当前指令在同一代码段内。如果加载的程序标号和当前指令不在同一代码段，汇编器将报错并中止汇编。另外在 Thumb 状态下，ADR 伪操作只能产生字对齐的地址加载指令。

ADRL 伪操作只能在 ARM 状态下使用。

ADR 和 ADRL 伪操作（ADR rn，label 或 ADRL rn，label）所能加载的地址范围依赖使用的指令集。

下面列出了在不同指令集下，ADR 伪操作所能加载的地址范围。

- ARM：在字节或半字节对齐的内存使用模式下，范围为±255 字节。在字对齐的内存使用模式下，范围为±1020 字节。
- 16 位 Thumb 指令集：0～1020 字节。Label 必须是字对齐地址标号，可以和伪操作 ALIGN 配合使用。
- 32 位 Thumb-2 指令集：±4095 字节（无论标号 label 是字、半字还是字节对齐）。

下面列出了在不同指令集下，ADRL 伪操作所能加载的地址范围。

- ARM：在字节或半字节对齐的内存使用模式下，范围为±64KB。在字对齐的内存使用模式下，范围为：±256KB。
- 16 位 Thumb 指令集：ADRL 伪指令不可用。
- 32 位 Thumb-2 指令集：±1M 字节（无论标号 label 是字、半字还是字节对齐）。

汇编过程中，汇编器将伪指令 ADR 编译成一条 ADD 或 SUB 指令，如果一条指令不能完成伪操作的功能，编译器将报错。ADRL 伪操作被编译器编译成两条数据处理指令，详细信息参见 ARM 伪操作一节。

下面的程序使用 ADR 伪操作加载了跳转表的入口地址，成功地实现了程序跳转。

```
        AREA    Jump, CODE, READONLY;将该子程序命名为 Name
        CODE32                      ;下面的代码为 ARM 代码
num     EQU     2                   ;要查找的跳转表入口号
```

```
        ENTRY                          ;程序入口
start                                  ;程序指令开始
        MOV    r0, #0                  ;为程序跳转加载 3 个参数
        MOV    r1, #3
        MOV    r2, #2
        BL     arithfunc               ;调用函数 arithfunc
stop    MOV    r0, #0x18               ;为 SWI 调用准备参数
        LDR    r1, =0x20026            ;调用 ADP_Stopped_ApplicationExit 功能
        SWI    0x123456                ;ARM semihosting SWI
arithfunc                              ;arithfunc 函数入口
        CMP    r0, #num                ;r0 中数值和 num 做比较
integer
        BXHS   lr                      ;如果 r0 ≥ num，函数返回
        ADR    r3, JumpTable           ;加载跳转表地址
        LDR    pc, [r3,r0,LSL#2]       ;跳转到相应的函数入口
JumpTable
        DCD    DoAdd
        DCD    DoSub
DoAdd   ADD    r0, r1, r2              ;r1 和 r2 相加，结果放入 r0
        BX     lr                      ;子函数返回
DoSub   SUB    r0, r1, r2              ;r1 和 r2 相减，结果放入 r0
        BX     lr                      ;子函数返回
        END                            ;程序结束
```

上面的程序段中，函数 arithfunc 带有 3 个参数（使用 r0、r1 和 r2 传参），函数的返回值通过 r0 返回。参数 1 决定该函数的功能。

- 参数 1=0，结果=参数 1+参数 2；
- 参数 1=1，结果=参数 1-参数 2。

程序中伪操作 LDR pc，[r3，r0，LSL#2]向 PC 寄存器加载了跳转表中的正确的子函数入口地址。

（2）使用 LDR Rd, = label 伪指令

使用 LDR Rd, = label 可以将 32 位的常数加载到寄存器。详见 ARM 伪指令一节。

ARM 汇编器首先将 label 地址存入数据缓冲池，在使用 LDR rn [pc，#offset to literal pool]指令将该 label 地址加载到寄存器。

使用 LDR Rd, = label 可以加载本段以外的标号 label 地址值（与 ADR 不同）。如果 LDR Rd, = label 加载的地址标号 label 和指令不在同一段，汇编器将在目标码中放置重定位伪操作指示连接器在连接时替换成合适地址值。

下面的例子使用 LDR Rd, = label 加载了本段之外的标号。

```
        AREA    LDRlabel, CODE,READONLY
        ENTRY                          ;程序入口
```

```
    start
           BL      func1               ;跳转到 func1
           BL      func2               ;跳转到 func2
    stop    MOV     r0, #0x18          ;为 SWI 调用准备参数
           LDR     r1, =0x20026        ;准备调用 SWI 的 ADP_Stopped_ApplicationExit 功
能
           SWI     0x123456            ;semihosting 软中断调用
    func1
           LDR     r0, =start          ;加载标号地址
           LDR     r1, =Darea + 12     ;该伪操作被编译为指令 LDR R1,[PC,#offset into
                                       ;Literal Pool 1]
           LDR     r2, =Darea + 6000   ;该伪操作被编译为指令 LDR R2, [PC, #offset into
                                       ;Literal Pool 1]
           MOV     pc,lr               ;程序返回
           LTORG                       ;内存池 1 入口
    func2
           LDR     r3, =Darea + 6000   ;编译为指令 LDR r3, [PC, #offset into
                                       ;Literal Pool 1]
                                       ;(共享内存池)
           ; LDR   r4, =Darea + 6004   ;从内存池 2 中加载新的数据，因为内存池 2 超出范围，
                                       ;所以指令失败
           BX      lr                  ;返回
    Darea   SPACE   8000               ;申请 8000 字节的内存空间并初始化为 0
           END                         ;程序结束，内存池 2 超出了 LDR 指令的数据加载范围
```

3．使用 LDM 和 STM 指令实现堆栈操作

无论 ARM、Thumb-2 还是 16 位 Thumb 指令集，都有相同的加载多个寄存器的指令（LDM 和 STM 指令）。多寄存器数据传输指令为寄存器和内存的多数据交换提供了有效方法。这些指令通常用于块拷贝或堆栈操作。使用多寄存器数据传输指令代替多条单寄存器传输指令的组合，有下面几点优势。

① 产生的代码量小，代码密度高。

② 在没有 Cache 的 ARM 处理器上使用多寄存器传输指令传输数据时，除第一个被传送的数据外，其余数据均是在连续的指令周期完成的（第一个被传送的数据使用非连续的内存周期），提高了指令的执行速度。

多寄存器的 LDM 和 STM 指令会增加中断的延时，因为 ARM 通常不会打断正在执行的指令去相应中断，而必须等到指令执行完。编译器会提供一个开关来控制 Load/Store 指令可以传送的最大寄存器数目以限制最大的中断延迟。

更多的关于 LDM 和 STM 指令的信息请参加 ARM 指令一节。

使用 ARM 汇编语言实现堆栈操作时，下面 4 条指令常被用到。

- LDM：加载多寄存器指令；
- STM：存储多寄存器指令；
- PUSH：将多个寄存器的值存储到堆栈并更新堆栈指针；
- POS：从堆栈中装载多个寄存器的值并更新堆栈指针。

下面分别介绍操作这些指令时的一些限制。

对于 LDM/STM 指令，可操作的寄存器要满足以下要求。

① ARM 状态下，r0～r15 中的任意寄存器或寄存器的组合。

② 在 32 位的 Thumb-2 指令集中，可以是 r0～r12 中的任意寄存器或寄存器的组合或有选择性的使用 r14 或 r15。

③ 在 16 位的 Thumb 指令集中，可以使用 r0～r7 中的任意寄存器或寄存器的组合。

对于 LDM/STM 指令，内存地址要满足以下要求。

① 数据传送后增长。

② 数据传送前增长（仅在 ARM 指令集中）。

③ 数据传送后减少（仅在 ARM 指令集中）。

④ 数据传送前减少（仅在 ARM 和 32 位 Thumb 指令集中）。

任何当前寄存器组的子集都可以使用多寄存器 LDM/STM 指令与寄存器进行数据交换。基址寄存器 Rn 决定目标或源地址，可以通过选择使用 Rn 后缀字符“!”来确定 Rn 的值是否随着传送而改变，就像使用回写前变址寻址的单寄存器传送指令一样。

对于 PUSH 和 POS，要满足以下要求。

① 使用堆栈指针寄存器 r13 作为基址寄存器，并在操作后更新该寄存器的值。

② 在每一个 POP 指令后，基址寄存器地址自动增加；在每一个 PUSH 指令后，基址寄存器地址自动减少。

③ 对指令操作的寄存器的限制同 LDM/STM 指令。

在 ARM 的 37 个寄存器中，R13 通常用作堆栈指针。堆栈寻址是隐含的，堆栈指针所指定的存储单元就是堆栈的栈顶，堆栈寻址通常有两种方式：向上生长（ascending）和向下生长（descending）。ARM 处理器有 ARM 和 Thumb 两种指令集。每种指令集都有丰富的指令可以对堆栈进行操作。堆栈指针指向最后压入堆栈的有效数据，称为满堆栈（full stack）；堆栈指针指向下一个数据项放入的空位置，称为空堆栈（empty stack）。根据堆栈的生长方向不同，可以生成 4 种类型的堆栈，即满递增、空递增、满递减和空递减。

注意 这种递增和递减的多寄存器传送指令不仅可以对堆栈进行操作，而且也可以用于正向或反向访问数组。

为方便堆栈的操作，ARM 指令增加了专门对堆栈操作的指令后缀。表 10.16 列出了对堆栈操作的数据传输指令。

表 10.16 对堆栈操作的数据传送指令

堆 栈 类 型	Push	Pop
满递减	STMFD（STMDB，Decrement Before）	LDMFD (LDM , Icrement Ater)
满递增	STMFA (STMIB, Increment Before)	LDMFA (LDMDA , Decrement After)

空递减	STMED (STMDA, Decrement After)	LDMED (LDMIB, Increment Before)
空递增	STMEA (STM , increment after)	LDMEA (LDMDB, Decrement Before)

下面的例子显示了数据传送指令对堆栈的操作。

```
STMFD r13!, {r0-r5}   ;r0～r5 入栈，该堆栈为满递减堆栈
LDMFD r13!, {r0-r5}   ;数据出栈，放入寄存器 r0～r5，该堆栈为满递减堆栈
```

注意 ARM 过程调用标准 AAPCS 定义使用满递减堆栈。使用 PUSH 和 POP 指令操作堆栈，堆栈类型默认为满递减堆栈，自动使用地址回写。

堆栈操作经常用于子程序调用的现场保护和子程序返回的现场恢复。另外，子程序的返回地址也常常被压栈保护。

下面的例子显示如何在子程序中进行现场保护和现场恢复。

```
subroutine PUSH {r5-r7,lr}   ;将工作寄存器和返回地址 lr 压栈
; code
BL somewhere_else
; code
POP {r5-r7,pc}               ;保存的寄存器数据和返回地址出栈
```

如果系统中存在 ARM 和 Thumb 的交互工作，以上代码只能用于 ARMv5 架构及其以上版本。在 ARMv4 架构中，直接将返回地址送 PC，不能引起处理器状态的切换。

下面的例子显示在符合 AAPCS 标准的满递减堆栈上，使用 STMFD 指令完成的 PUSH 操作（见图 10.1）。STMFD 指令把寄存器内容压栈，SP 指针指向栈顶。

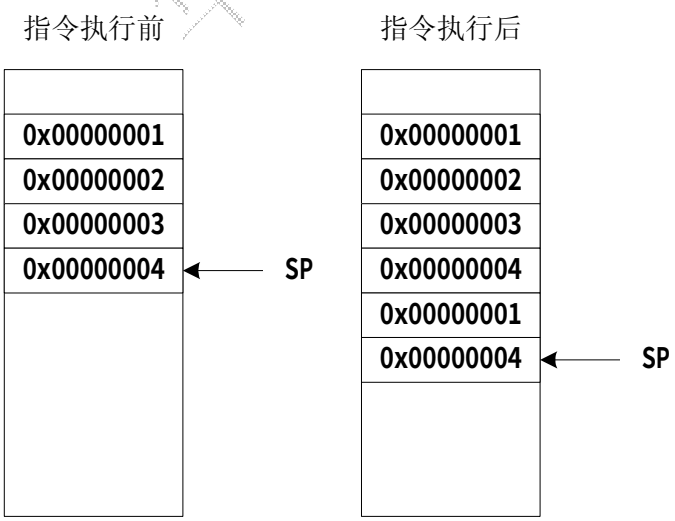


图 10.1　满递减堆栈的 STMFD 操作

```
        MOV  r1，0x01
```

```
        MOV  r4，0x04
        STMFD  SP!，{r1，r4}
```

若要检查一个堆栈，必须关注堆栈的 3 个属性：堆栈基址、堆栈指针及堆栈限制。堆栈基址是堆栈在存储器中的起始地址；堆栈指针初始时指向堆栈基址单元，随着数据压栈，堆栈指针连续移动并始终指向栈顶；如果堆栈指针超过了堆栈限制，就会发生堆栈溢出错误。下面代码用于检测递减式堆栈的溢出错误。

```
    SUB  SP，SP，#size
    CPM  SP，r10
    BLLO  _stack_overflow      ；条件
```

AAPCS 把寄存器 r10 定义为堆栈限制或 sl（stack limit）寄存器。这是一个可选的操作，因为，堆栈检查只有在堆栈检查使能的时候才可以使用。BLL0 指令是一个附加了条件助记符 L0 的带链接的分支指令。如果在执行一个 push 操作后，SP 的值小于 r10 的值，就发生了堆栈溢出错误。如果堆栈指针在执行 pop 操作后，超出了堆栈基址，那么就产生了堆栈下溢（stack underflow）错误。

4．使用 LDM 和 STM 指令实现块复制

当程序中有大量数据需要复制或搬移时，常用到 LDM 和 STM 指令。虽然使用单寄存器数据传送指令 LDR 和 STR 也能实现同样的功能，但代码密度和执行效率要低于使用多寄存器传送指令。

下面通过两个例子说明了使用单寄存器数据传送指令和使用多寄存器数据传送指令的区别。

```
        AREA   Word, CODE, READONLY    ;给代码段起名为 Word
    num EQU    20                      ;num=20 将要拷贝的字的个数
        ENTRY                          ;程序入口
    call
    start
        LDR    r0, =src     ;r0 = 源操作块起始地址
        LDR    r1, =dst     ;r1 = 目的操作块起始地址
        MOV    r2, #num     ;r2 = 将要拷贝的字的个数
    wordcopyLDR    r3, [r0], #4 ;从源操作块装载一个字
        STR    r3, [r1], #4 ;存储到目的操作块
        SUBS   r2, r2, #1   ;指针移到下一个要拷贝的字单元
        BNE    wordcopy     ;循环拷贝
    stop    MOV    r0, #0x18    ;angel_SWIreason_ReportException 为软中断调用准备
参数
        LDR    r1, =0x20026 ;调用 Semihosting 的 ADP_Stopped_ApplicationExit 功
能
```

```
            SWI    0x123456     ;ARM semihosting SWI 调用
            AREA   BlockData, DATA, READWRITE
    src     DCD    1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
    dst     DCD    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
            END
```

下面是使用多寄存器数据拷贝命令 LDM 和 STM 重写的代码。

```
            AREA  Block, CODE, READONLY        ;给代码段起名为 Block
    num     EQU   20              ;num=20 将要拷贝的字的个数
            ENTRY                 ;程序入口
    call
    start
            LDR   r0, =src        ;r0 = 源操作块起始地址
            LDR   r1, =dst        ;r1 = 目的操作块起始地址
            MOV   r2, #num        ;r2 = 将要拷贝的字的个数
            MOV   sp, #0x400      ;设置堆栈指针 (r13)
    blockcopy MOVS  r3,r2, LSR #3 ;判断要移动的字的个数是 8 的几倍
            BEQ   copywords       ;如果移动的字的个数小于 8 则跳转到 copywords 子函数
            PUSH  {r4-r11}        ;将用到的工作寄存器压栈保存
    octcopy   LDM   r0!, {r4-r11} ;从源地址拷贝 8 个字
            STM   r1!, {r4-r11}   ;存到目的地址
            SUBS  r3, r3, #1      ;计数器减 1
            BNE   octcopy         ;如果计数器不等于零，继续拷贝
            POP   {r4-r11}        ;如果计数器等于零，恢复工作寄存器
    copywords ANDS  r2, r2, #7    ;判断要拷贝的剩余字个数
            BEQ   stop            ;判断剩余字数是否为零
    wordcopy  LDR   r3, [r0], #4  ;从源地址加载一个字
            STR   r3, [r1], #4    ;到目的地址
            SUBS  r2, r2, #1      ;计算器减 1
            BNE   wordcopy      ;如果计数器不等于零，继续拷贝
    stop      MOV   r0, #0x18     ;为软中断调用准备参数
            LDR   r1, =0x20026 ;调用 Semihosting 的 ADP_Stopped_ApplicationExit
功能
            SWI   0x123456      ;调用 Semihosting 软中断
            AREA  BlockData, DATA, READWRITE
    src     DCD   1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
    dst     DCD   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
            END
```

为了增加数据拷贝的效率，程序中使用了“MOVS r3,r2, LSR #3”指令。该指令将为下面以 8 个字节为单位进行数据拷贝做准备。将数据以 8 个字为单位拷贝，是因为 ARM 体

系结构中所能使用的寄存器为 8 个，即：r4～r11（根据 AAPCS 标准，r0～r3 在子程序调用过程中将被使用）。

10.5.3 宏的使用

使用伪操作 MACRO 和 MEND 可以将一段代码定义为宏。程序运行时，使用宏名调用宏。程序中使用宏的优势在于。

① 提高代码的可读性，使用更能代表程序功能的名称代替特定的代码段。

② 避免同一代码段在程序中重复多次。

1．宏在分支程序中的应用

在 ARMv6T2 之前的 ARM 体系结构中，一个“测试－分支”操作需要至少两条 ARM 指令完成。可以使用宏定义这种“分支－测试”结构。

```
        MACRO
$label  TestAndBranch  $dest, $reg, $cc
$label  CMP     $reg, #0
         B$cc   $dest
         MEND
```

上面的程序段定义了一个叫做 TestAndBranch 的宏，并且为该宏定义了 4 个参数（$label、$dest、$reg 和$cc），在程序中使用该宏时，必须为参数赋值。下面的程序段显示了如何调用 TestAndBranch 宏。

```
test    TestAndBranch   NonZero, r0, NE
          ...
          ...
NonZero
After substitution this becomes:
test    CMP     r0, #0
         BNE    NonZero
          ...
          ...
NonZero
```

2．宏在除法中的应用

下面的例子程序定义的宏实现了无符号整数的除法，调用该宏需要为以下 4 个参数赋值。

- $Bot：保存除法除数的寄存器。
- $Top：指令执行前保存被除数，指令执行后保存除数。
- $Div：保存除法运算的商，如果除法只要求余数，该寄存器的值为 0。
- $Temp：临时寄存器。

```
        MACRO
$Lab    DivMod  $Div,$Top,$Bot,$Temp
        ASSERT  $Top <> $Bot            ;如果使用的寄存器不全相等，产生错误信息
        ASSERT  $Top <> $Temp
        ASSERT  $Bot <> $Temp
        IF      "$Div" <> ""
           ASSERT  $Div <> $Top         ;如果$Div 不为空，产生三个提示信息
           ASSERT  $Div <> $Bot
           ASSERT  $Div <> $Temp
        ENDIF
$Lab
        MOV     $Temp, $Bot             ;将除数放入$Temp
        CMP     $Temp, $Top, LSR #1     ;$Temp 乘以 2，直到 2*$Temp>$Top
90      MOVLS   $Temp, $Temp, LSL #1
        CMP     $Temp, $Top, LSR #1
        BLS     %b90                    ;b 表示向后搜索
        IF      "$Div" <> ""            ;如果$Div 为空，忽略下一条指令
           MOV     $Div, #0             ;将$Div 初始化
        ENDIF
91      CMP     $Top, $Temp             ;Can we subtract $Temp?$Top 大于$Temp?
        SUBCS   $Top, $Top,$Temp        ;如果大于，$Top 减去$Temp，结果放入$Top
        IF      "$Div" <> ""            ;如果$Div 为空，忽略下一条指令
           ADC     $Div, $Div, $Div     ;将$Div 乘以 2
        ENDIF
        MOV     $Temp, $Temp, LSR #1    ;将$Temp 除 2
        CMP     $Temp, $Bot             ;循环，直到$Temp 小于除数
        BHS     %b91
        MEND
```

下面是该除法宏被引用时的结果。

```
        ASSERT   r5 <> r4           ;如果寄存器全不相等，显示提示信息
        ASSERT   r5 <> r2
        ASSERT   r4 <> r2
        ASSERT   r0 <> r5           ;如果$Div 不为空，产生三个提示信息
        ASSERT   r0 <> r4
        ASSERT   r0 <> r2
ratio
        MOV      r2, r4             ;将除数放在 r2 中
        CMP      r2, r5, LSR #1     ;r2 乘以 2，直到 2*r2>r5
```

```
90      MOVLS   r2, r2, LSL #1
        CMP     r2, r5, LSR #1
        BLS     %b90            ;b 表示向后搜索
        MOV     r0, #0          ;初始化 r0
91      CMP     r5, r2          ;判断 r5 是否大于 r2
        SUBCS   r5, r5, r2      ;如果大于，r5 减去 r2，结果放入 r5
        ADC     r0, r0, r0      ;r0 乘以 2
        MOV     r2, r2, LSR #1  ;r2 除以 2
        CMP     r2, r4          ;循环，直到 r2<r4
        BHS     %b91
```

10.5.4 使用 MAP 和 FIELD 命令描述数据结构

可以使用 MAP 和 FIELD 伪操作来描述数据结构。这两个伪操作一般是一起使用的。

使用 MAP 和 FIELD 伪操作定义数据结构有以下好处。

- 容易维护。
- 可以方便地实现相同数据结构的重复定义。
- 可以更高效地存取数据。

MAP 伪操作指定数据结构的基址。

FIELD 伪操作指定一个数据项所需的存储器数量，并为该数据项指定一个标号。对结构中的每个数据项重复该命令。

注意 使用 MAP 和 FIELD 伪操作当定义一个数据结构时，不分配存储器空间。使用定义常数的命令（如 DCD）来分配存储器空间。

1．相对映射

MAP/FIELD 伪操作和使用寄存器相对寻址的 LOAD/STORE 指令配合使用，访问事先定义好的结构体是 MAP/FIELD 伪操作的基本用法。

下面是一个使用寄存器相对寻址的 LOAD 指令访问结构体的例子。

```
MAP 0
consta FIELD 4          ;consta 占用 4 字节，偏移量为 0
constb FIELD 4          ;constb 占用 4 字节，偏移量为 4
x FIELD 8               ;x 占用 8 字节，偏移量为 8
y FIELD 8               ;y 占用 8 字节，偏移量为 16
string FIELD 256        ;string 占用 256，偏移量为 24
```

上面定义的数据结构，可以使用下列指令来访问：

```
MOV r9,#4096
LDR r4,[r9,#constb]
```

标号是相对于数据结构的开始位置的。用于存放映射的起始地址的寄存器（此例中为 r9）称为基址寄存器。

数据结构的位置是由运行时装载到基址寄存器的值确定的。MAP/FIELD 伪操作不实际分配内存地址。

同一映射可以用于描述数据结构的多个实例。它们可以位于存储器中的任何位置。

备注 r9 是“ARM-Thumb 程序调用标准”中的静态基址寄存器 (sb)。详细信息请参阅本书附录。

2．基于寄存器的映射

一般情况下，每次访问一个数据结构时可以使用相同的寄存器作为基址寄存器。可以在使用 MAP 定义映射的基址时指定该寄存器的名称。

下面的例子显示了一个基于寄存器的映射。

```
    MAP 0,r9
    consta FIELD 4            ;consta 变量占用 4 个字节，地址偏移量为 0（相对于 r9 中的地址）
    constb FIELD 4            ;constb 变量占用 4 个字节，地址偏移量为 4
    x FIELD 8                 ;x 占用 8 个字节，地址偏移量为 8
    y FIELD 8                 ;y 占用 8 个字节，地址偏移量为 16
    string FIELD 256          ;字符串占用 256 字节，起始地址偏移量为 24
```

利用示例中的映射，可以访问数据结构（不管它在内存中什么位置）：

```
    ADR r9,datastart
    LDR r4,constb ; => LDR r4,[r9,#4]       ;该伪操作被编译为 LDR r4,[r9,#4]
```

constb 包含从数据结构开始位置算起的数据项的偏移量，也包含基址寄存器。在此例中，基址寄存器是在 MAP 命令中定义的 r9。

3．相对于程序的映射

可以使用程序计数器 (r15) 作为一个映射的基址寄存器。当使用 r15 做为基址寄存器时，被使用 LOAD/STORE 指令寻址的数据结构偏移量不能超出 4KB 范围。这是因为使用 PC 寄存器为基址的 LOAD/STORE 指令，最大寻址能力为 4KB。

下面的例子显示了使用 r15 作为基址寄存器的程序段。其中包含一个为数据结构分配存储器空间的 SPACE 伪操作。

```
    datastruc SPACE 280     ;保留 280 个字节来定义数据结构
    MAP datastruc
    consta FIELD 4
    constb FIELD 4
```

```
x FIELD 8
y FIELD 8
string FIELD 256
```

使用下面的指令读取 constb 域所包含的内容。

```
LDR r2,constb     ; => LDR r2,[pc,offset]     该伪操作被编译为 LDR r2,[pc,offset]
```

在此例中，不需要在装载数据之前装载基址寄存器，此时使用程序计数器 PC 作为默认寄存器。

注意 由于处理器里面的流水线，r15 寄存器值和 LDR 指令的实际地址并不相同。但是，汇编器将修正此问题。

4．定义结构体的结束

可以使用带有 0 操作数的 FIELD 伪操作来标记结构内的一个位置。标记位置后位置计数器并未增加（即偏移量 offset 不增加）。

下面的例子使用事先定义好的常量 MaxStrLen 和 ArrayLen 定义了字符串和数组的大小。为了防止 MaxStrLen 和 ArrayLen 值过大，使结构体超出所能使用的内存范围，使用了“FIELD 0”伪操作来标识结构体的结束。

```
StartOfData EQU 0x1000
EndOfData EQU 0x2000
MAP StartOfData
Integer FIELD 4
Integer2 FIELD 4
String FIELD MaxStrLen
Array FIELD ArrayLen*8
BitMask FIELD 4
EndOfUsedData FIELD 0
ASSERT EndOfUsedData <= EndOfData
```

该例中使用了：

- 一个 EQU 命令定义可使用的最大内存空间；
- 一个带有 0 操作数的 FIELD 伪操作来标记数据结构的结束位置；
- 一个 ASSERT 伪操作确认数据结构的结束位置并未超出可用内存。

5．强制内存对齐

如果在定义数据结构时，使用了一些字符变量（或其他非 4 字节的变量），就可能造成内存访问的对齐问题。

下面例子中显示了一个数据结构的定义。该定义方式使整数访问不能保证在字边界对齐。

```
StartOfData EQU 0x1000
```

```
EndOfData EQU 0x2000
MAP StartOfData
Char FIELD 1
Char2 FIELD 1
Char3 FIELD 1
Integer FIELD 4    ; 非字边界对齐
Integer2 FIELD 4
String FIELD MaxStrLen
Array FIELD ArrayLen*8
BitMask FIELD 4
EndOfUsedData FIELD 0
ASSERT EndOfUsedData <= EndOfData
```

这里，不能使用 ALIGN 伪操作来修正该边界对齐问题，因为 ALIGN 伪操作只对齐存储器内的当前位置。而 MAP 和 FIELD 伪操作不为其定义的结构分配任何存储空间。

解决的方法是，在 Char3 FIELD 1 后插入一个虚拟的 FIELD 1。但是，如果改变了字符变量的数目，就会造成维护困难。每次必须重新计算正确的填充数。

下面的例子说明了一个更好的调整填充的方法。该示例使用一个带有 0 操作数的 FIELD 伪操作来标记字符数据的结束位置。第二个 FIELD 命令根据标号的值插入正确的填充数。使用“:AND:”运算符来计算正确的值。

```
StartOfData EQU 0x1000
EndOfData EQU 0x2000
MAP StartOfData
Char FIELD 1
Char2 FIELD 1
Char3 FIELD 1
EndOfChars FIELD 0
Padding FIELD (-EndOfChars):AND:3
Integer FIELD 4
Integer2 FIELD 4
String FIELD MaxStrLen
Array FIELD ArrayLen*8
BitMask FIELD 4
EndOfUsedData FIELD 0
ASSERT EndOfUsedData <= EndOfData
```

“(-EndOfChars):AND:3”表达式计算正确的填充数：

- 如果 EndOfChars 是 0 mod 4，则填充数是 0；
- 如果 EndOfChars 是 1 mod 4，则填充数是 3；

- 如果 EndOfChars 是 2 mod 4，则填充数是 2；
- 如果 EndOfChars 是 3 mod 4，则填充数是 1。

无论何时添加或删除字符变量，它会自动调整填充数。

华清远见

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 11 章 Thumb 指令集

在 ARM 体系结构中，ARM 指令集中的指令是 32 位指令，其执行效率很高。对于存储系统数据总线为 16 位的应用系统，ARM 体系提供了 Thumb 指令集。Thumb 指令集是 ARM 指令集的一个子集，它比 ARM 指令集有更高的代码密度（一个可执行的程序在内存中所占的空间）。在存储系统受限的嵌入式系统中，比如移动电话、PDA 等，代码密度是非常重要的，同时，成本压力也会限制存储器的大小、数据宽度和速度。在 ARM 体系的 T 变种（T variable）的版本中，同时支持 ARM 指令集和 Thumb 指令集，而且遵循一定的调用规则时，Thumb 子程序和 ARM 子程序可以相互调用。

11.1 Thumb 指令的特点及实现

Thumb 指令集把 32 位 ARM 指令集的一个子集编码为一个 16 位的指令集。在 16 位外部数据总线宽度下，ARM 处理器上使用 Thumb 指令的性能要比使用 ARM 指令的性能更好；而在 32 位外部数据总线宽度下，使用 Thumb 指令的性能要比使用 ARM 指令的性能差。因此，Thumb 指令多用于存储器受限的一些系统中。Thumb 指令集并没有改变 ARM 系统底层的程序设计模型，只是在该模型上增加了一些限制条件。Thumb 指令集中的数据处理指令的操作数仍然是 32 位，指令寻址地址也是 32 位的。

代码密度是 Thumb 指令集的一个主要优势。平均而言，对于同一个程序，使用 Thumb 指令实现所需的存储空间，要比等效的 ARM 指令实现少 30%左右。下面的例子代码，使用 ARM 指令和 Thumb 指令实现相同的除法操作。从例子中可以看出，虽然 Thumb 指令的实现使用了更多的指令，但是它占用的总的存储空间却比较小。

【例 11.1】使用 ARM 指令实现除法运算

```
        MOV  r3，＃0
loop
        SUB  r0，r0，r1
        ADDGE  r3，r3，＃1
        BGE  loop
        ADD  r2，r0，r1
```

【例 11.1】中 r0 为被除数，r1 存放除数，r2 和 r3 分别存放余数和商。完成整个除法运算使用了 5 条指令，每一条指令所占的字节数为 4，所以实现一个除法运算，ARM 指令所占有的字节数为 20。

【例 11.2】使用 Thumb 指令实现除法运算

```
        MOV  r3，＃0
loop
        ADD  r3，＃1
        SUB  r0，r1
        BGE  loop
        SUB  r3，＃1
        ADD  r2，r0，r1
```

【例 11.2】使用 Thumb 指令完成了和【例 11.1】完全相同的功能。Thumb 指令虽然使用了 6 条指令，但其每条指令占用 2 个字节，所以总的字节数为 6×2=12，小于 ARM 指令所占用的 20 个字节。

Thumb 指令是 ARM 指令的一个受限子集，在 Thumb 状态下，不能直接访问所有的处理器寄存器，只有 r0～r7 是可以被任意访问的，在 Thumb 状态下使用该 8 个寄存器和在 ARM 状态下使用没有区别。寄存器 r8～r12 只能通过 MOV、ADD 或 CMP 指令访问。CMP 指令和所有操作 r0～r7 的数据处理指令都会影响 CPSR 中的条件标志位。一些 Thumb 指令还使用

到了程序计数器 PC（r15），链接地址寄存器 LR（r14）和堆栈指针寄存器 SP（r13）。在 Thumb 状态下，读取 r15 寄存器时，bit[0]值为 0，bit [31：1]包含了 PC 的值。当对 r15 进行写入时，bit[0]被忽略，bit[31：1]被设置成当前程序计数器的值。

表 11.1 列出了 Thumb 状态下，各寄存器的使用情况。

表 11.1　　Thumb 寄存器的使用

寄　存　器	访　　问
r0～r7	完全访问
r8～r12	只能通过 MOV、ADD 及 CMP 访问
r13	限制访问
r14	限制访问
r15	限制访问
CPSR	间接访问
SPSR	不能访问

从表 11.1 可以看出，Thumb 状态下不能直接访问 CPSR 和 SPSR。也就是没有和 MSR 和 MRS 等价的指令。为了改变 CPSR 和 SPSR 的值，必须使处理器状态切换到 ARM 状态，再使用指令 MSR 和 MRS 来实现。同样，在 Thumb 状态下也没有协处理器访问指令，要访问协处理器寄存器来配置 cache 和进行内存管理，也必须使处理器切换到 ARM 状态。

注意　Thumb 状态下，对 CPSR 的条件标准位控制由算术和逻辑操作设置并控制条件转移。

11.2　Thumb 编程模型

所有的 Thumb 指令都是 16 位的。它们都是 ARM 指令重新编码得到的，所以继承了 ARM 指令集的许多特点。

① 有数据处理、数据传送和流控制的指令结构。

② 支持 8 位字节、16 位半字和 32 位字数据类型，半字以两字节边界对齐，字以 4 字节边界对齐。

③ 32 位的无分段存储器（unsegmented memory）。

Thumb 指令集除了继承了 ARM 指令集的一些特点外，与 ARM 指令集存在以下一些差异。

① 大多数 Thumb 指令为无条件执行指令（所有 ARM 指令都是条件执行的）。

② 许多 Thumb 数据处理指令采用了 2 地址格式（目的寄存器与源寄存器相同）。而 ARM 指令中除 64 位乘法指令外，其余指令均采用 3 地址模式。

③ Thumb 指令格式减少了很多 ARM 指令格式的限制，使 Thumb 指令编写的代码密度

大大提高。

无论处理器处于什么状态，所有的异常都使处理器返回到 ARM 状态，并完成异常处理。但异常发生时，CPSR 状态寄存器在进入异常时被保存到相应的 SPSR 中，当异常处理结束后，处理器将恢复到异常发生前的状态，并按照发生异常时处理器的状态继续执行 ARM 或 Thumb 指令。

应该注意的是，ARM 异常返回指令需要根据 ARM 流水线的行为对返回地址进行调整。由于 Thumb 指令是 2 字节长，而 ARM 指令是 4 字节长，所以由 Thumb 执行状态进入异常时其自然偏移量应与 ARM 不同（ARM 状态下，拷贝到链接寄存器的值 PC－4）。为了减少编程的复杂性，ARM 体系结构中设置了硬件逻辑，以实现 Thumb 状态的自动地址偏移调整，使 ARM 和 Thumb 状态编程一致。

表 11.2 列出了 Thumb 状态下发生异常时 LR 的值。

表 11.2　　Thumb 状态异常返回指令

异　　常	异常链接寄存器值	返 回 指 令
Reset	不可预知的值	—
未定义指令	未定义指令地址+2	MOV　PC，r14
SWI	Swi 指令地址+2	MOV　PC，r14
预取异常	预取异常指令+4	MOV　PC，r14，＃4
数据异常	产生预取异常指令地址+8	MOV　PC，r14，＃8
IRQ	下一条将被执行的指令地址+4	MOV　PC，r14，＃4
FIQ	下一条将被执行的指令地址+4	MOV　PC，r14，＃4

11.3　Thumb 跳转指令

Thumb 指令集中的跳转指令分以下 6 种类型。

① 无条件跳转，其跳转空间为±2KB。

② 条件跳转，其跳转空间为±256B。

③ 带返回的跳转指令，其跳转空间为±4MB。

④ 带状态切换的跳转指令（是否进行状态切换可以在程序中设定选择）。

⑤ 带返回和状态切换的跳转指令（是否进行状态切换可以在程序中设定选择）。

⑥ 第二种形式的带返回和状态切换的跳转指令。

下面详细介绍各指令的特点及用途。

11.3.1　跳转指令 B

Thumb 中有两个分支跳转指令的变体，第一个变体与 ARM 版本指令相似，可条件执行，跳转被限制在有符号 8 位立即数所表示的范围内，或者是±256B。第二个变体不可条件执行（没

有条件码部分),但扩展了有效跳转范围,跳转范围为有符号 11 位立即数表示的范围,即±2048B。

条件分支指令是 Thumb 指令中惟一可以条件执行的指令。

首先来介绍可条件执行的跳转指令 B（1）。

（1）编码格式

可条件执行的跳转指令的编码格式如图 11.1 所示。

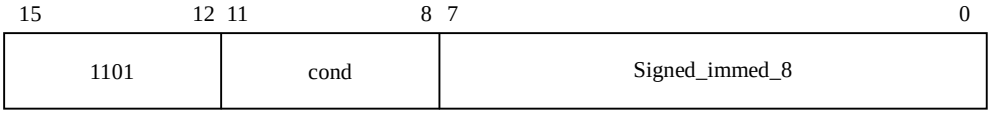


图 11.1　B（1）指令编码格式

（2）指令的语法格式

```
B<cond>  <target_address>
```

① <cond>

指定指令的执行条件。条件助记符与 ARM 中相同。

② <target_address>

指定程序跳转的目标地址。指令通过下面的方法计算目标地址。

- 将指令中包含的 8 位有符号数左移一位。
- 将结果符号扩展为 32 位。
- 将得到的值加到 PC 寄存器中，即得到跳转的目标地址。

有条件的跳转指令可以实现±256B 范围的程序跳转。

（3）指令操作的伪代码

```
if  ConditionPass{cond}  then
    PC=PC+{SignExtend(Signed_immed_8)<<1}
```

（4）指令的使用

为了得到正确的 signed_immed_8，汇编器需要执行以下的操作步骤。

① 首先形成跳转的基地址。该跳转的基地址是跳转指令地址加 4。也就是说，跳转指令的基地址即当前程序指针寄存器的值。

② 从跳转的目标地址中减去基地址形成跳转偏移量。该偏移量应为偶数（因为 Thumb 指令为半字对齐）。

③ 如果跳转偏移量超出-256～+254B 范围，汇编器产生一个错误。

④ 将产生的跳转偏移量除以 2 放入指令编码中的 signed_immed_8 域。

注意　如果该指令的条件域为 AL，即指令编码条件域为 0b1110 时，程序产生未定义指令异常。当指令的条件域为 NV，即指令编码条件域为 0b1111 时，指令等价于 SWI 指令。

（5）ARM 指令集中的跳转指令

该指令与 ARM 指令集中 B<cond> <target_address>基本相似，所不同的是 ARM 指令集中，偏移量左移两位而 Thumb 指令集中偏移量左移一位。另外，处理器在 ARM 和 Thumb

状态下所读取的 PC 值也是不同的。

下面介绍无条件跳转指令 B（2）。

（1）编码格式

无条件执行的跳转指令的编码格式如图 11.1 所示。

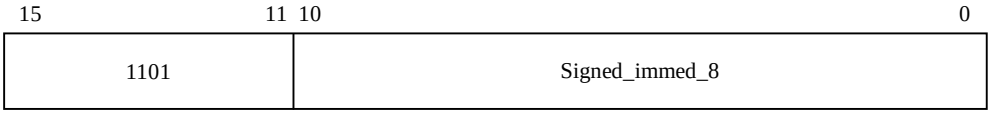


图 11.2　B（2）指令编码格式

（2）指令的语法格式

```
B  <target_address>
```

<target_address>

指定程序跳转的目标地址。指令通过下面的方法计算目标地址。

- 将指令中包含的 11 位有符号数左移一位。
- 将结果符号扩展为 32 位。
- 将得到的值加到 PC 寄存器中，即得到跳转的目标地址。

有条件的跳转指令可以实现±2048B 范围的程序跳转。

（3）指令操作的伪代码

```
PC=PC+{SignExtend(Signed_immed_8)<<1}
```

（4）指令的使用

为了得到正确的 signed_immed_11，汇编器需要执行以下的操作步骤。

① 首先形成跳转的基地址。该跳转的基地址是跳转指令地址加 4。也就是说，跳转指令的基地址即当前程序指针寄存器的值。

② 从跳转的目标地址中减去基地址形成跳转偏移量。该偏移量应为偶数（因为 Thumb 指令为半字对齐）。

③ 如果跳转偏移量超出-2048～+2046B 范围，汇编器产生一个错误。

④ 将产生的跳转偏移量除以 2 放入指令编码中的 signed_immed_11 域。

（5）ARM 指令集中的跳转指令

该指令与 ARM 指令集中 B <target_address>基本相似，所不同的是 ARM 指令集中偏移量左移两位，而 Thumb 指令集中偏移量左移一位。另外，处理器在 ARM 和 Thumb 状态下所读取的 PC 值也是不同的。

11.3.2　带返回的无条件跳转指令 BL

（1）编码格式

带返回的无条件跳转指令的编码格式如图 11.3 所示。

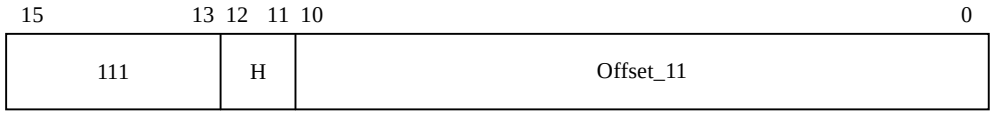


图 11.3　BL 指令编码格式

带返回的跳转指令 BL 提供了一种在 Thumb 状态下程序间相互调用的方法，当从子程序返回时，通常使用下面的方式之一：

- MOV　PC，LR
- BX　LR
- POP　{pc}

BL 指令不可条件执行，可以实现在大约±4MB 的地址空间范围内跳转，实现方法是将一条 BL 指令编译成两条 16 位的 Thumb 指令，从而实现上述跳转。对编译后的两条指令说明如下：

① H=10 的 BL 指令。该跳转包含跳转偏移量的高位部分。

② H=11 的 BL 指令。该跳转包含跳转偏移量的低位部分。

（2）指令的语法格式

```
BL  <target_address>
```

<target_address>

指定程序跳转的目标地址。指令通过下面的方法计算目标地址。

- 将 H=10 的 BL 指令的 offset_11 域左移 12 位。
- 将结果符号扩展为 32 位。
- 将得到的值加到 PC 寄存器中。
- 与 H=11 的 BL 指令的 offset_11 域相加。

因此 BL 指令可以实现在大约±4MB 的地址空间范围内跳转。

（3）指令操作的伪代码

```
if  H==10  then
    LR=PC+(SignExtend(offset_11)<<12)
Else  if  H==11  then
    PC=LR+(offset_11<<11)
    LR=(address  of  next  instruction)|1
Else  if  H==01  then
    PC=(LR+(offset_11<<1)) AND 0xFFFFFFFC
    LR=(address of next instruction)|1
Else  if  H==01  then
    PC=(LR+(offset_11<<1)) AND 0Xfffffffc
    LR=(address of next instruction)|1
    T Flag=0
```

（4）指令的使用

为了能够正确产生两条 Thumb 跳转指令，汇编器按照如下步骤产生跳转偏移量。

① 形成跳转基地址。此基地址为 H=10 时的 BL 指令地址加上 4，即执行该条指令的 PC 值。

② 从目标地址中减去基地址，形成跳转偏移量。

根据以上步骤所产生的结果是-2^{22}～$+2^{22}-2$ 之间的一个偶数，如果结果超出此范围，汇编器将报错。

③ 如果产生的结果在给定范围内，汇编器将产生下面两条 BL 指令：

- H=10，offset_11=offset[22：12]
- H=11，offset_11=offset[11：1]

注意 当 H=00 时，该指令为无条件跳转指令。

（5）ARM 指令集中的 BL 指令

如果调用 Thumb 子程序，该指令类似于 BLX <target_addr>；如果程序调用 ARM 子程序，该指令类似于 BL <target_addr>。

11.3.3 带返回链接的无条件跳转指令 BLX（1）

（1）编码格式

带返回的无条件跳转指令的编码格式如图 11.4 所示。



图 11.4 BLX（1）指令编码格式

带返回链接的跳转指令 BLX（1）提供了一种在 Thumb 状态下无条件调用 ARM 子程序的方法，当从子程序返回时，通常使用下面的方式之一：

- BX LR;
- 加载 PC 的 LDR 或 LDM 指令。

BLX 指令不可条件执行，可以实现在大约±4MB 的地址空间范围内跳转，实现方法是将一条 BLX 指令编译成两条 16 位的 Thumb 指令，从而实现上述跳转。对编译后的两条指令说明如下：

① H=10 的跳转指令。该跳转包含跳转偏移量的高位部分。

② H=01 的跳转指令。该跳转包含跳转偏移量的低位部分。

（2）指令的语法格式

```
BLX <target_address>
```

① <target_address>

指定程序跳转的目标地址。指令通过下面的方法计算目标地址。

- 将 H=10 的 BL 指令的 offset_11 域左移 12 位。
- 将结果符号扩展为 32 位。

- 将得到的值加到 PC 寄存器中。
- 与 H=11 的 BL 指令的 offset_11 域相加。

BL 指令可以实现在±4MB 的地址空间范围内跳转。

（3）指令操作的伪代码

```
if  H==10  then
    LR=PC+(SignExtend(offset_11)<<12)
Else  if  H==11  then
    PC=LR+(offset_11<<11)
    LR=(address  of  next  instruction)|1
Else  if  H==01  then
    PC=(LR+(offset_11<<1)) AND 0xFFFFFFFC
    LR=(address of next instruction)|1
Else  if  H==01  then
    PC=(LR+(offset_11<<1)) AND 0Xfffffffc
    LR=(address of next instruction)|1
    T Flag=0
```

（4）指令的使用

为了能够正确产生两条 Thumb 的跳转指令，汇编器按照如下步骤产生跳转偏移量。

① 形成跳转基地址。此地址为 H=10 时地址加上 4，即执行该条指令的 PC 值。

② 使基址地址的 bit[1]等于目标地址的 bit[1]（保证 ARM 状态的字地址对齐）。

③ 从目标地址中减去基地址，形成跳转偏移量。

根据以上步骤所产生的结果是-2^{22}～$+2^{22}-2$ 之间的一个偶数，如果结果超出此范围，汇编器将报错。

④ 如果产生的结果在给定范围内，汇编器将产生下面两条 BL 指令：

- H==10，offset_11=offset[22：12]
- H==01，offset_11=offset[11：1]

（5）等效 ARM 指令

该指令类似于 ARM 指令集的 BL　<target_addr>。

11.3.4　带状态切换的跳转指令 BX

（1）编码格式

带状态切换的跳转指令 BX 的编码格式如图 11.5 所示。

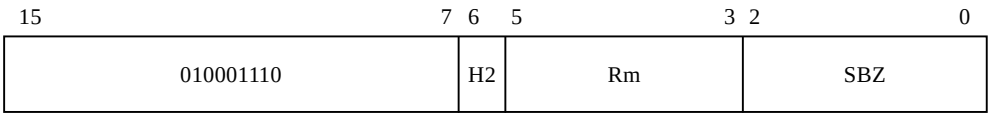


图 11.5　BX 指令的编码格式

BX 指令用于 ARM 和 Thumb 程序之间的调用。

（2）指令的语法格式

```
BX  <Rm>
```

其中<Rm>为目标地址寄存器，包含程序的跳转地址。BX 指令的目标地址寄存器可以是 r0～r15 中的任意寄存器。

注意 如果 Rm[1: 0]=0b10，不满足 ARM 指令的内存对齐方式。指令的执行结果不可预知。如果该指令使用 r15 作为目标寄存器，其操作方式和使用其他寄存器相同。

（3）指令操作的伪代码

```
T  Flag=Rm[0]
PC=Rm[31: 1]<<1
```

（4）ARM 指令集中的 BX 指令

ARM 指令集中的 BX 指令和 Thumb 指令集中的 BX 指令相差较大，它们分别为不同方向的跳转。当 r15 作为目的寄存器使用时，要特别注意该指令在两个指令集中的区别。

11.3.5 带返回链接的无条件跳转指令 BLX（2）

（1）编码格式

带返回链接的无条件跳转指令 BLX（2）的编码格式如图 11.6 所示。

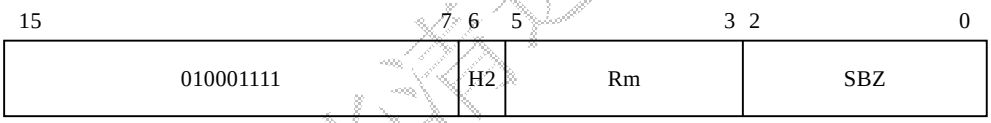


图 11.6 BLX（2）指令的编码格式

该 BLX（2）指令用于 ARM 和 Thumb 子程序间的相互调用。程序状态字的 T 标志位根据目的寄存器的 bit[0]位而改变。

（2）指令的语法格式

```
BLX  <Rm>
```

其中<Rm>为目标地址寄存器，r0～r14 寄存器均可以做为目标地址寄存器。

注意 如果在此指令中使用 r15 作为目的寄存器，指令的执行结果不可预知。

此指令只在 ARMv5 版本以上指令集中被支持。

（3）指令操作的伪代码

```
LR = (address of the instruction after this BLX)|1
T Flag = Rm[0]
PC = Rm[31:1]<<1
```

11.3.6 Thumb 指令集中跳转指令举例

下面的例子程序综合使用了各种跳转指令，通过该例可以对 Thumb 状态下程序的非连续执行有更深入的了解。

```
        B  label            ;无条件跳转到 Label 地址
        BCC label           ;如果进位标志为 0，则跳转
        BEQ label           ;如果零标准位置 1，则跳转
        BL     func         ;子程序调用
    Func
        …
        …                   ;子程序体
        MOV PC，LR          ;子程序返回
        BX      r12         ;跳转到 r12 寄存器中保存的地址
```

在执行此例中的 BX　r12 时，如果 r12 的 bit[0]=0，则处理器进入 ARM 状态执行，否则继续执行 Thumb 代码。

11.4 Thumb 数据处理指令

数据处理指令是指那些操作寄存器中数据的指令。Thumb 指令集中的数据处理指令是 ARM 指令集数据处理指令的一个子集，其中包括 MOV 指令、算术指令、移位指令、逻辑指令、比较指令和乘法指令。表 11.3 列出了 Thumb 数据处理指令。

表 11.3　　Thumb 状态数据处理指令

助　记　符	说　　明	操　　作
ADC　Rd,Rm	带进位的 32 位加	Rd:=Rd+Rm+C flag
ADD　Rd,Rn,Rm	32 位加	Rd:=Rn+Rm
ADD　Rd,Rn,#0~#7	32 位加	Rd:=Rn+3_bit_immed
ADD　Rd,#0~#277	32 位加	Rd:=Rn+8_bit_immed
AND　Rd,Rm	逻辑与	Rd:=Rd AND Rm
ASR　Rd,Rm,#1~#32	算术右移	Rd:=Rm ASR 5_bit_immed
ASR　Rd,Rs	算术右移	Rd:=Rm ASR Rs

续表

助　记　符	说　　明	操　　作
BIC　Rd,Rm	位清零	Rd:=Rd AND NOT Rm
CMN　Rn,Rm	32 位取负比较	Rn+Rm　并设置标志位

CMP Rn,#0~#255	32 位整数比较	Rn-8_bit_immed 并设置标志位
CMP Rn,Rm	32 位整数比较	Rn-Rm 并设置标志位
EOR Rd,Rm	异或	Rd:=Rd EOR Rm
LSL Rd,Rm,#0~#31	逻辑左移	Rd:=Rm LSL 5_bit_immed
LSL Rd,Rs	逻辑左移	Rd:=Rd LSL Rs
LSR Rd,Rm,#1~#32	逻辑右移	Rd:=Rm LSR 5_bit_immed
LSR Rd,Rs	逻辑右移	Rd:=Rd LSR Rs
MOV Rd,#0~#255	将数据送入寄存器	Rd:=8_bit_immed
MOV Rd,Rn	将数据送入寄存器	Rd:=Rn
MUL Rd,Rm	乘	Rd:=Rm*Rd
MVN Rd,Rm	将 32 位数的“反”送入寄存器	Rd:=NOT Rm
NEG Rd,Rm	求反	Rd:=0-Rm
ORR Rd,Rm	逻辑或	Rd:=Rd OR Rm
ROR Rd,Rs	逻辑右移	Rd:=Rd ROR Rs
SBC Rd,Rm	带进位减	Rd:=Rd-Rm-NOT(Carry Flag)
SUB Rd,Rn,Rm	减	Rd:=Rn-Rm
SUB Rd,Rn,#0~#7	减	Rd:=Rn-3_bit_immed
SUB Rd,#0~#255	减	Rd:=Rn-8_bit_immed
TST Rn,Rm	位测试指令	Rn AND Rm 并更新标志位

Thumb 的数据处理指令与等价的 ARM 指令使用相同的格式。所有对 r0～r7 低 8 个寄存器操作的数据处理指令都更新条件标志位，对 r8～r14 和 PC 高 8 个寄存器操作的指令除 MOV 指令外，其他指令均不改变条件标志位。这些指令包括：

- MOV Rd，Rn
- ADD Rd，Rm
- CMP Rn，Rm
- ADD sp，＃0～＃508
- SUB sp，＃0～＃508
- ADD Rd，sp，＃0～＃1020
- ADD Rd，pc，＃0～＃1020

Thumb 数据处理指令的基本语法格式分为以下 8 种。

① <opcode1> <Rd>，<Rn>，<Rm>
<opcode1>: =ADD|SUB
② <opcode2> <Rd>，<Rn>，#<3_bit_immed>
<opcode2>: =ADD|SUB

```
③ <opcode3>  <Rd>|<Rn>，#<8_bit_immed>
   <opcode3>: =ADD|SUB|MOV|CMP
④ <opcode4>  <Rd>，<Rm>，#<shift_immed>
   <opcode4>: =LSL|LSR|ASR
⑤ <opcode5>  <Rd>|<Rn>，<Rm>|<Rs>
   <opcode5>: =MVN|CMP|CMN|TST|ADC|SBC|NEG|MUL|LSL|LSR|ASR|ROR|AND|EOR|BIC
⑥ ADD  <Rd>，<reg>，#<8_bit_immed>
   <reg>: =SP|PC
⑦ <opcode6>  SP，SP，#<7_bit_immed>
   <opcode6>: =ADD|SUB
⑧ <opcode7>  <Rd>|<Rn>，<Rm>
   <opcode7>: =MOV|ADD|CMP
```

注意 上面的指令和语法格式中，3_bit_immed、7_bit_immed、8_bit_immed 分别表示 3 位、7 位、8 位立即数。

下面详细介绍各指令的语法和使用。

11.4.1 ADC 指令

（1）编码格式

带进位的加法指令 ADC 的编码格式如图 11.7 所示。

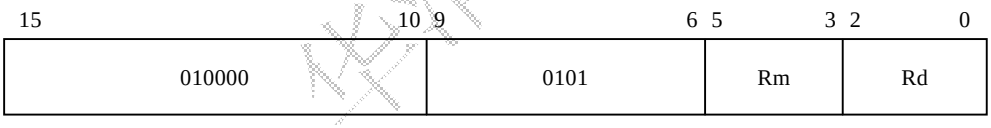


图 11.7　ADC 指令的编码格式

带进位的 ADC 指令和加法指令 ADD 配合使用可以实现多字相加（multi-word），详见 ARM 指令集介绍。

（2）指令的语法格式

```
ADC  <Rd>，<Rm>
```

① <Rd>

第一个操作数寄存器，并且是操作结果的目的寄存器。

② <Rm>

第二操作数寄存器。

（3）指令操作的伪代码

```
Rd = Rd+Rm+C Flag
N Flag = Rd[31]
Z Flag = if Rd==0 then 1 else 0
```

```
C Flag = CarryFrom{Rd+Rm+C Flag}
V Flag = OverflowFrom{Rd+Rm+C Flag}
```

（4）对应的 ARM 指令

```
ADCS  <Rd>，<Rd>，<Rm>
```

11.4.2 小立即数加法指令 ADD（1）

（1）编码格式

立即数加法指令 ADD（1）的编码格式如图 11.8 所示。

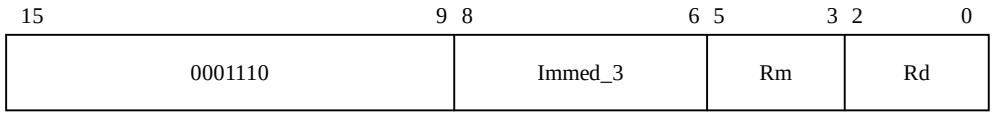


图 11.8 ADD（1）指令的编码格式

ADD 加法指令实现“小常数”的加法运算，并将相加的结果保存到指定的寄存器中。

（2）指令的语法格式

```
ADD  <Rd>，<Rn>， #<immed_3>
```

① <Rd>

加法操作的目标寄存器。

② <Rn>

操作数寄存器。

③ #<immed_3>

3 位的立即数。该立即数将和寄存器<Rn>的值相加，并将结果保存到<Rd>中。

（3）指令操作的伪代码

```
Rd = Rn+immed_3
N Flag = Rd[31]
Z Flag = if Rd==0 then 1 else 0
C Flag = CarryFrom{Rn+ immed_3}
V Flag = OverflowFrom{Rn+ immed_3}
```

（4）对应的 ARM 指令

```
ADDS  <Rd>，<Rn>， #<immed_3>
```

11.4.3 大立即数加法指令 ADD（2）

（1）编码格式

立即数加法指令 ADD（2）的编码格式如图 11.9 所示。

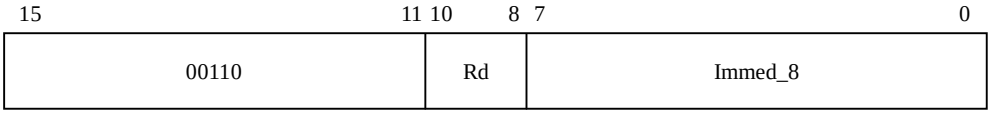


图 11.9　ADD（2）指令的编码格式

此中形式的 ADD（2）指令将一个大的立即数（8 bit）与指定寄存器的值相加，并将运算结果保存到此寄存器中。

（2）指令的语法格式

```
ADD  <Rd>，＃<immed_8>
```

① <Rd>

加法操作的目标寄存器。

② ＃<immed_8>

8 位的立即数。该立即数将和寄存器<Rd>的值相加，并将结果保存到<Rd>中。

（3）指令操作的伪代码

```
Rd = Rd+immed_8
N Flag = Rd[31]
Z Flag = if Rd==0 then 1 else 0
C Flag = CarryFrom{Rn+ immed_8}
V Flag = OverflowFrom{Rn+ immed_8}
```

（4）对应的 ARM 指令

```
ADDS  <Rd>，<Rd>，＃<immed_8>
```

11.4.4　寄存器加法指令 ADD（3）

（1）编码格式

寄存器加法指令 ADD（3）的编码格式如图 11.10 所示。

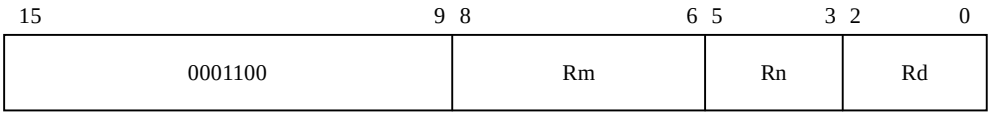


图 11.10　ADD（3）指令的编码格式

此种形式的加法指令将两个寄存器的值相加，将结果放入第三个目标寄存器，并根据操作结果更新标志位。

（2）指令的语法格式

```
ADD  <Rd>，<Rn>，<Rm>
```

① <Rd>

加法操作的目标寄存器。

② <Rn>

操作数寄存器。存放加法操作的第一个操作数。

③ <Rm>

操作数寄存器。存放加法操作的第二个操作数。

（3）指令操作的伪代码

```
Rd=Rn+Rm
N Flag = Rd[31]
Z Flag = if Rd = =0 then 1 else 0
C Flag = CarryFrom(Rn+Rm)
V Flag = OverflowFrom(Rn+Rm)
```

11.4.5 寄存器加法指令 ADD（4）

（1）编码格式

寄存器加法指令 ADD（4）的编码格式如图 11.11 所示。

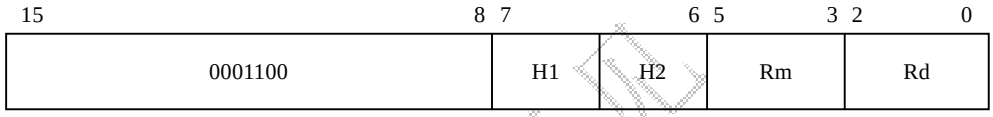


图 11.11 ADD（4）指令的编码格式

此种形式的加法指令将两个寄存器的值相加，将结果放入第三个目标寄存器。该指令不更新程序状态字的标志位。

注意 该指令与 ADD（3）的区别在于其操作数寄存器。该指令的操作数寄存器为 r8～r14 和 PC 高寄存器。操作结果对程序状态字的标志位没有影响。

（2）指令的语法格式

```
ADD  <Rd>，<Rm>
```

① <Rd>

指令的操作数寄存器，其中保存加法操作的一个操作数，并将指令的操作结果放入该寄存器。取值范围为 r0～r15。

② <Rm>

操作数寄存器。存放加法操作的第二个操作数。可以为 r0～r15 的任意寄存器。

（3）指令操作的伪代码

```
Rd = Rd+Rm
```

11.4.6 PC 相关加法指令 ADD（5）

（1）编码格式

寄存器加法指令 ADD（5）的编码格式如图 11.12 所示。

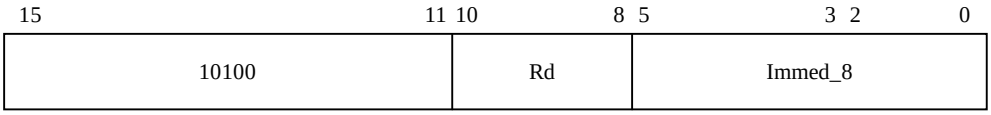


图 11.12　ADD（5）指令的编码格式

该指令将一个立即数和 PC 值相加，并将 PC 相关地址写入目标寄存器。立即数可以是 0～1020 范围内的任意数值的 4 倍。该指令不更新程序状态字的标志位。

（2）指令的语法格式

```
ADD  <Rd>, PC, #<immed_8>×4
```

① <Rd>

指令的目的寄存器，存放指令的操作结果。

② PC

PC 相关地址。

③ <immed_8>

加到 PC 值上的 8 位立即数。

（3）指令操作的伪代码

```
Rd=（PC AND 0xfffffffc）+（immed_8<<2）
```

11.4.7　SP 相关加法指令 ADD（6）

（1）编码格式

寄存器加法指令 ADD（6）的编码格式如图 11.13 所示。

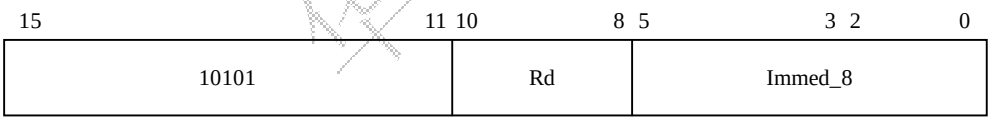


图 11.13　ADD（6）指令的编码格式

该指令将一个立即数和 SP 值相加，并将 SP 相关地址写入目标寄存器。立即数可以是 0～1020 范围内的任意 4 的倍数。该指令不更新程序状态字的标志位。

（2）指令的语法格式

```
ADD  <Rd>, SP, #<immed_8>×4
```

① <Rd>

指令的目的寄存器，存放指令的操作结果。

② SP

SP 相关地址。

③ <immed_8>

该立即数的 4 倍将与 SP 值相加。

（3）指令操作的伪代码

```
Rd=SP+（immed_8<<2）
```

11.4.8 SP 相关加法指令 ADD（7）

（1）编码格式

寄存器加法指令 ADD（6）的编码格式如图 11.14 所示。

该指令将一个立即数和 SP 值相加，并将 SP 相关地址写回 SP 寄存器。立即数可以是 0～508 范围内的任意数值的 4 倍。该指令不更新程序状态字的标志位。

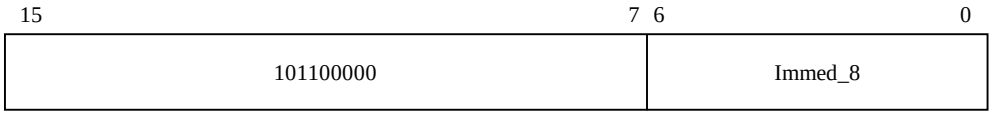


图 11.14 ADD（7）指令的编码格式

（2）指令的语法格式

```
ADD SP, #<immed_7>×4
```

① SP

SP 相关地址，同时也为指令的目标寄存器。

② <immed_7>

指定的 7 位立即数，该立即数的 4 倍将与 SP 值相加。

11.4.9 逻辑与指令 AND

（1）编码格式

逻辑与指令 AND 的编码格式如图 11.15 所示。

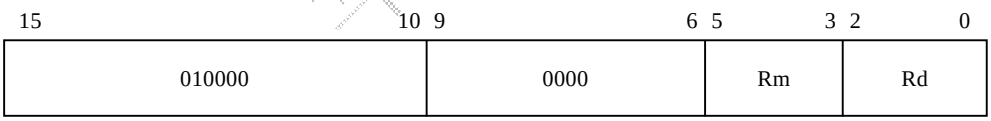


图 11.15 AND 指令的编码格式

AND 指令实现两个寄存器值的按位“与”操作。程序状态字的标志位根据指令的执行结果更新。

（2）指令的语法格式

```
AND <Rd>, <Rm>
```

① <Rd>

操作数寄存器，包含指令的第一个操作数。同时也为指令操作结果的目的寄存器。

② <Rm>

操作数寄存器，保护指令的第二个操作数。

（3）指令操作的伪代码

```
Rd = Rd AND Rm
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

（4）对应的 ARM 指令

```
ANDS  <Rd>，<Rd>，<Rm>
```

11.4.10　算术右移指令 ASR（1）

（1）编码格式

算术右移指令 ASR（1）的编码格式如图 11.16 所示。

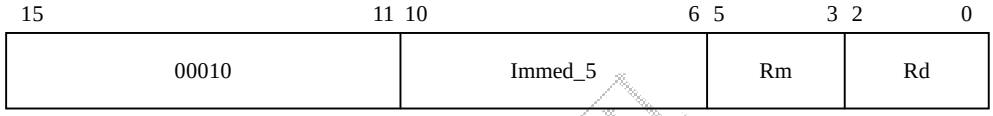


图 11.16　ASR（1）指令的编码格式

这种形式的算术右移指令可以方便的实现将一个寄存器的值除以一个常数。该常数是以 2 为底的幂。

（2）指令的语法格式

```
ASR  <Rd>，<Rm>，＃<immed_5>
```

① <Rd>

目的寄存器。用于存放指令操作的结果。

② <Rm>

操作数寄存器。存放将要被右移的数据。

③ <immed_5>

指定右移的位数。该常数取值范围为 1～31。

（3）指令操作的伪代码

```
If  immed_5 == 0
     C Flag = Rm[31]
     If Rm[31] = =0 then
         Rd = 0
     Else /*Rm[31] = =1*/
         Rd = 0xffffffff
Else /*immed_5 > 0*/
     C Flag = Rm[immed_5-1]
```

```
        Rd = Rm Arithmetic_shift_Right immed_5
    N Flag = Rd[31]
    Z Flag = if Rd = =0 then 1 else 0
    V Flag = unaffected
```

（4）对应的 ARM 指令

```
MOVS  <Rd>，<Rm>，ASR#<immed_5>
```

注意 在 ARMv5 以前的体系结构版本中，没有单独的 Thumb 移位指令。在 ARMv6 版本中已经增加了位移指令，详细信息请参加 ARM 体系结构相关文档。

11.4.11 算术右移指令 ASR（2）

（1）编码格式

算术右移指令 ASR（2）的编码格式如图 11.17 所示。

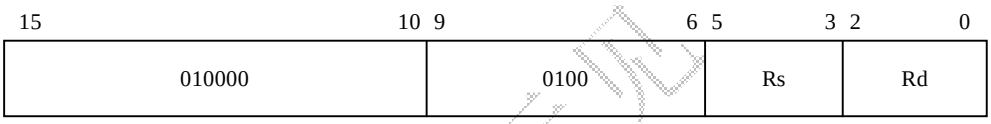


图 11.17 ASR（2）指令的编码格式

此种形式的 ASR 指令的操作数均为寄存器。该指令根据指令的操作结果更新程序状态字的标志位。

（2）指令的语法格式

```
ASR  <Rd>，<Rs>
```

① <Rd>

存放指令的操作数和操作结果。

② <Rs>

指定操作数将要被移动的位数。

（3）指令操作的伪代码

```
If  Rs[7:0] = = 0 then
    C Flag = unaffected
    Rd = unaffected
Else if Rs[7:0] < 32 then
    C Flag = Rd[Rs[7:0] - 1]
    Rd = Rd arithmetic_shift_Right Rs[7:0]
Else /*Rs[7:0] >= 32*/
    C Flag = Rd[31]
    If Rd[31] = = 0 then
```

```
        Rd = 0
    Else /*Rd[31] = = 1*/
        Rd = 0xffffffff
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
V Flag = unaffected
```

（4）对应的 ARM 指令

```
MOVS  <Rd>，<Rd>，ASR<Rs>
```

11.4.12　位清零指令 BIC

（1）编码格式

位清零指令 BIC 的编码格式如图 11.18 所示。

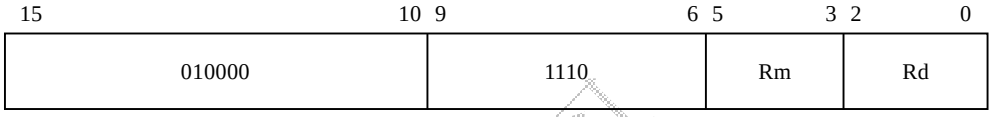


图 11.18　BIC 指令的编码格式

BIC 指令将两个寄存器的值按位做“异或”操作。该指令根据指令的执行结果更新程序状态字的标志位。

（2）指令的语法格式

```
BIC  <Rd>，<Rm>
```

① <Rd>

存放指令的操作数和操作结果。

② <Rm>

操作数寄存器，该寄存器中的数据的反码将会和<Rd>中的值做“与”操作。

（3）指令操作的伪代码

```
Rd = Rd AND NOT Rm
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

（4）对应的 ARM 指令

```
BICS  <Rd>，<Rd>，<Rm>
```

11.4.13　比较指令 CMN

（1）编码格式

比较指令 CMN 的编码格式如图 11.19 所示。

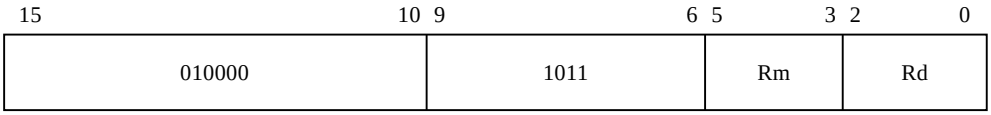


图 11.19　CMN 指令的编码格式

比较指令 CMN 将一个寄存器的值和另一个寄存器的值取负做比较，并根据比较结果更新程序状态字的标志位。程序中比较指令后通常跟条件执行指令。

（2）指令的语法格式

```
CMN  <Rn>，<Rm>
```

① <Rn>

操作数寄存器。用于存放指令的第一个操作数。

② <Rm>

操作数寄存器。用于存放指令的第二个操作数。

（3）指令操作的伪代码

```
Alu_out = Rn + Rm
N Flag = alu_out[31]
Z Flag = if alu_out = = 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn + Rm)
V Flag = OverflowFrom（Rn+Rm）
```

（4）对应的 ARM 指令

```
CMN  <Rn>，<Rm>
```

11.4.14　比较指令 CMP（1）

（1）编码格式

比较指令 CMP（1）的编码格式如图 11.20 所示。

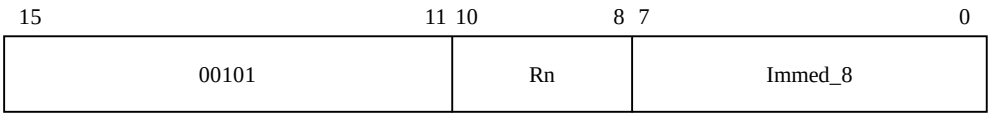


图 11.20　CMP（1）指令的编码格式

比较指令 CMP（1）将一个寄存器的值和 8 位立即数做比较，并根据比较结果更新程序状态字的标志位。程序中比较指令后通常跟条件执行指令。

（2）指令的语法格式

```
CMP  <Rn>, #<immed_8>
```

① <Rn>
操作数寄存器。
② <immed_8>
8 位常数。将与寄存器<Rn>的值做比较。
（3）指令操作的伪代码

```
Alu_out = Rn – immed_8
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn – immed_8)
V Flag = OverflowFrom(Rn – immed_8)
```

（4）对应的 ARM 指令

```
CMP  <Rn>, #<immed_8>
```

11.4.15　比较指令 CMP（2）

（1）编码格式
比较指令 CMP（2）的编码格式如图 11.21 所示。

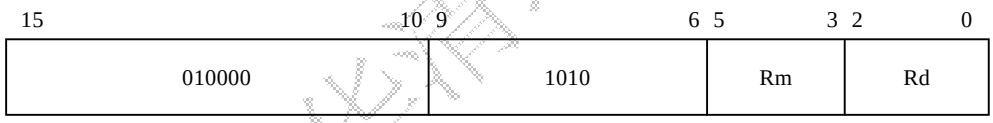


图 11.21　CMP（2）指令的编码格式

比较指令 CMP（2）将两个寄存器的值做比较，并根据比较结果更新程序状态字的标志位。程序中比较指令后通常跟条件执行指令。
（2）指令的语法格式

```
CMP  <Rn>, <Rm>
```

① <Rn>
操作数寄存器，存放比较的第一个操作数。
② <Rm>
操作数寄存器，存放比较的第二个操作数。
（3）指令操作的伪代码

```
alu_out = Rn – Rm
N Flag = alu_out[31]
Z Flag = if alu_out = = 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - Rm)
```

```
V Flag = OverflowFrom(Rn - Rm)
```

（4）对应的 ARM 指令

```
CMP  <Rn>，<Rm>
```

11.4.16 比较指令 CMP（3）

（1）编码格式

比较指令 CMP（3）的编码格式如图 11.22 所示。

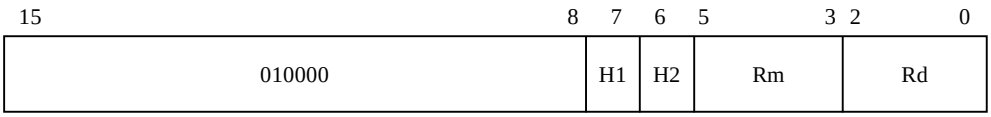


图 11.22 CMP（3）指令的编码格式

比较指令 CMP（3）将两个寄存器的值做比较，其中一个寄存器为 r8～r14 或 PC 寄存器高 16 位。并根据比较结果更新程序状态寄存器的标志位。程序中比较指令后通常跟条件执行指令，实现指令的分支跳转。

（2）指令的语法格式

```
CMP  <Rn>，<Rm>
```

① <Rn>

操作数寄存器。保存将要进行比较的第一个操作数，其取值可以为 r0～r15 的任意寄存器。

② <Rm>

操作数寄存器。保存将要进行比较的第二个操作数，其取值可以为 r0～r15 的任意寄存器。

（3）指令操作的伪代码

```
alu_out = Rn – Rm
N Flag = alu_out[31]
Z Flag = if alu_out = = 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - Rm)
V Flag = OverflowFrom (Rn - Rm)
```

注意 指令操作数寄存器<Rn>和<Rm>中，至少有一个寄存器为 r8～r14 或 PC 寄存器的高 16 位，否则，指令的执行结果不可预知。

（4）对应的 ARM 指令

```
CMP  <Rn>，<Rm>
```

11.4.17 异或指令 EOR

（1）编码格式

比较指令 EOR 的编码格式如图 11.23 所示。

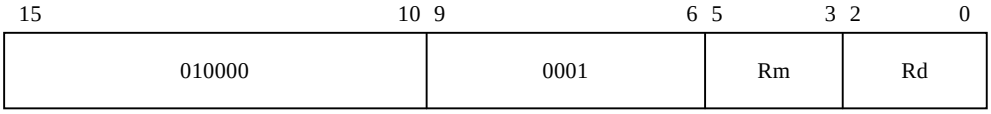


图 11.23　EOR 指令的编码格式

EOR 指令将两个寄存器中的数值按位执行“异或”操作，并根据指令的执行结果更新程序状态寄存器的标志位。

（2）指令的语法格式

```
EOR  <Rn>，<Rm>
```

① <Rn>

操作数寄存器。同时保存指令的第一个操作数和指令的执行结果。

② <Rm>

操作数寄存器。保存将要进行比较的第二个操作数。

（3）指令操作的伪代码

```
Rd = Rd OR Rm
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

（4）对应的 ARM 指令

```
EORS  <Rd>，<Rd>，<Rm>
```

11.4.18　逻辑左移指令 LSL（1）

（1）编码格式

逻辑左移指令 LSL（1）的编码格式如图 11.24 所示。

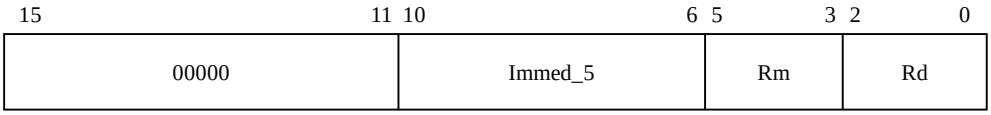


图 11.24　LSL（1）指令的编码格式

逻辑左移指令 LSL（1）可以实现以 2 为底的幂的乘法。进行移位后空出的位添 0。

（2）指令的语法格式

```
LSL  <Rd>，<Rm>，#<immed_5>
```

① <Rd>

目的寄存器。存储指令的操作结果。

② <Rm>

操作数寄存器。该寄存器保存的数据将进行左移操作。

③ <immed_5>

逻辑左移位数，范围为 0～31。

（3）指令操作的伪代码

```
if  immed_5 = = 0
     C Flag = unaffected
     Rd = Rm
Else  /*immed_5 > 0*/
     C Flag =Rm[32 – immed_5]
     Rd = Rm logical_shift_left immed_5
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
V Flag = unaffected
```

（4）对应的 ARM 指令

```
MOVS  <Rd>，<Rm>，LSL # <immed_5>
```

11.4.19 逻辑左移指令 LSL（2）

（1）编码格式

逻辑左移指令 LSL（2）的编码格式如图 11.25 所示。

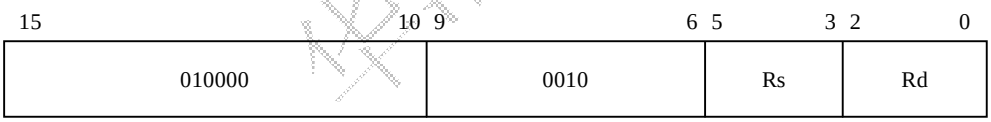


图 11.25 LSL（2）指令的编码格式

逻辑左移指令 LSL（2）可以实现以 2 为底的幂的乘法。进行移位后空出的位添 0 并根据指令的操作结果更新程序状态寄存器的标志位。

（2）指令的语法格式

```
LSL  <Rd>，<Rs>
```

① <Rd>

操作数寄存器，包含被移位的值并保存指令的执行结果。

② <Rs>

包含逻辑左移位数的寄存器。

（3）指令操作的伪代码

```
if  Rs[7:0] = = 0
    C Flag = unaffected
```

```
    Rd = unaffected
Else  if  Rs[7:0] < 32 then
    C Flag = Rd[32 – Rs[7:0]]
    Rd = Rd logical_shift_left  Rs[7:0]
Else  if  Rs[7:0] = = 32 then
    C Flag = Rd[0]
    Rd = 0
Else  if  Rs[7:0] = = 32 then
    C Flag = Rd[0]
    Rd = 0
Else  /*Rs[7:0] > 32*/
    C Flag = 0
    Rd = 0
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
V Flag = unaffected
```

（4）对应的 ARM 指令

```
MOVS  <Rd>，<Rd>，LSL<Rs>
```

11.4.20　逻辑右移指令 LSR（1）

（1）编码格式

逻辑左移指令 LSR（1）的编码格式如图 11.26 所示。

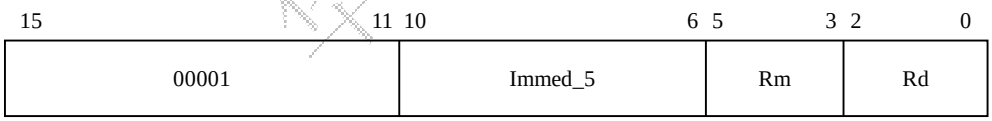


图 11.26　LSR(1)指令的编码格式

逻辑右移指令 LSR（1）可以实现以 2 为底的幂做除数的除法。进行移位后空出的位添 0，并根据指令的执行结果更新程序状态寄存器的标志位。

（2）指令的语法格式

```
LSR  <Rd>，<Rm>，＃<immed_5>
```

① <Rd>

目的寄存器。存储指令的操作结果。

② <Rm>

操作数寄存器。该寄存器保存的数据将进行右移操作。

③ <immed_5>

逻辑右移位数，范围为 0～31。

（3）指令操作的伪代码

```
if  immed_5 = = 0
     C Flag = Rd[31]
     Rd = 0
Else /*immed_5>0*/
     C Flag = Rd[immed_5 - 1]
     Rd = Rm Logial_shift_right immed_5
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
V Flag = unaffected
```

（4）对应的 ARM 指令

```
MOVS  <Rd>，<Rm>，LSR#<immed_5>
```

11.4.21 逻辑右移指令 LSR（2）

（1）编码格式

逻辑左移指令 LSR（2）的编码格式如图 11.27 所示。

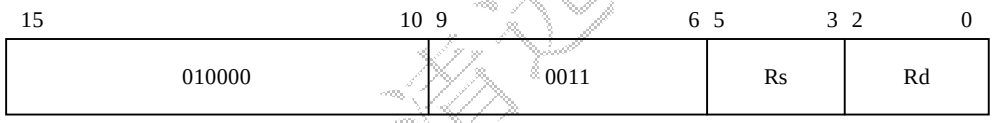


图 11.27 LSR(2)指令的编码格式

逻辑右移指令 LSR（2）可以实现以 2 为底的幂做除数的无符号除法。进行移位后空出的位添 0，并根据指令的操作结果更新程序状态寄存器的标志位。

（2）指令的语法格式

```
LSR  <Rd>，<Rs>
```

① <Rd>

操作数寄存器，包含被移位的值并保存指令的执行结果。

② <Rs>

包含逻辑右移位数的寄存器。

（3）指令操作的伪代码

```
if  Rs[7:0] = = 0
    C Flag = unaffected
    Rd = unaffected
Else  if  Rs[7:0] < 32 then
    C Flag = Rd[Rs[7:0] - 1]
```

```
    Rd = Rd logical_shift_Right  Rs[7:0]
Else  if  Rs[7:0] = = 32 then
    C Flag = Rd[31]
    Rd = 0
Else  if  Rs[7:0] = = 32 then
    C Flag = Rd[0]
    Rd = 0
Else  /*Rs[7:0] > 32*/
    C Flag = 0
    Rd = 0
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
V Flag = unaffected
```

（4）对应的 ARM 指令

```
MOVS  <Rd>，<Rd>，LSR<Rs>
```

11.4.22　移位指令 MOV（1）

（1）编码格式

数据传送指令 MOV（1）的编码格式如图 11.28 所示。

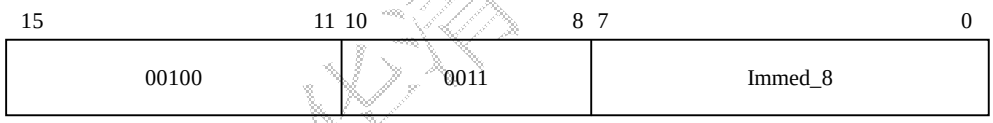


图 11.28　MOV(1)指令的编码格式

（2）指令的语法格式

```
MOV  <Rd>，#<immed_8>
```

① <Rd>

目的寄存器。存放指令的执行结果。

② <immed_8>

8 位立即数。指示要移入寄存器的数据。

（3）指令操作的伪代码

```
Rd = immed_8
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

（4）对应的 ARM 指令

```
MOVS  <Rd>, #<immed_8>
```

11.4.23 移位指令 MOV（2）

（1）编码格式

数据传送指令 MOV（2）的编码格式如图 11.29 所示。

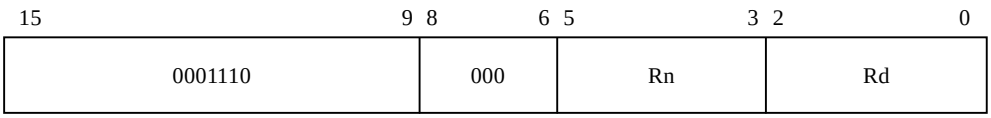


图 11.29 MOV（2）指令的编码格式

此种形式的 MOV 指令用于数据在 r0～r7 低寄存器间传送，并根据的指令的执行结果更新程序状态寄存器的标志位。

注意 从指令的编码形式可以看出，MOV（2）指令被作为 ADD Rd，Rn，#0 指令来编译。

（2）指令的语法格式

```
MOV  <Rd>, <Rn>
```

① <Rd>

目的寄存器。存放指令的执行结果。

② <Rn>

源寄存器。用于存放被传送的数据。

（3）指令操作的伪代码

```
Rd = Rn
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = 0
V Flag = 0
```

（4）对应的 ARM 指令

```
ADD  <Rd>, <Rn>, #0
```

11.4.24 移位指令 MOV（3）

（1）编码格式

数据传送指令 MOV（3）的编码格式如图 11.30 所示。

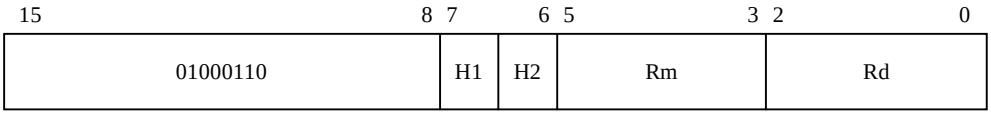


图 11.30　MOV（3）指令的编码格式

此种形式的 MOV 指令用于在 r8～r14 和 PC 寄存器的高 16 位内传送数据。与 MOV（2）指令不同，该指令不影响程序状态寄存器。

（2）指令的语法格式

```
MOV  <Rd>，<Rm>
```

① <Rd>

目的寄存器。可以是 r0～r15 中的任意寄存器，用于存放指令的执行结果。

② <Rm>

源寄存器。可以是 r0～r15 中的任意寄存器，用于存放被传输的数据。

（3）指令操作的伪代码

```
Rd = Rm
```

（4）指令的使用

可以使用此种形式的 MOV 指令 MOV　PC，r14 用于子程序的返回（返回的程序是 Thumb 子程序）。它和 BX 指令的区别在于 BX 指令可以用于 ARM 和 Thumb 程序间的相互调用。

注意　如果目的寄存器<Rd>和源寄存器<Rm>同时为 r0～r7 低寄存器，则指令的执行结果不可预知。

（5）对应的 ARM 指令

```
MOV  <Rd>，<Rm>
```

11.4.25　乘法指令 MUL

（1）编码格式

乘法指令 MUL 的编码格式如图 11.31 所示。

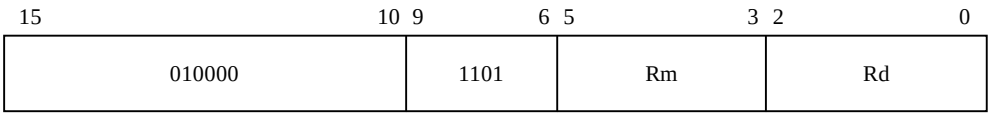


图 11.31　MUL 指令的编码格式

MUL 指令实现两个数（可以为无符号数，也可以为有符号数）的乘积，并将指令的执行结果存放到一个 32 位的寄存器中，同时可以根据运算结果设置 CPSR 寄存器中相应的条件标志位。

（2）指令的语法格式

```
MUL  <Rd>，<Rm>
```

① <Rd>

目的寄存器，存放乘法操作的第一个乘数。

② <Rm>

第二个乘数所在寄存器。

（3）指令操作的伪代码

```
Rd = (Rm * Rd)[31:0]
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

注意 如果乘法指令 MUL 的源和目的寄存器使用相同的寄存器，则指令的执行结果不可预知；对于有符号数和无符号数，指令的操作结果是一样的。

（4）对应的 ARM 指令

```
MULS  <Rd>，<Rm>，<Rd>
```

11.4.26 传送指令 MVN

（1）编码格式

传送指令 MVN 的编码格式如图 11.32 所示。

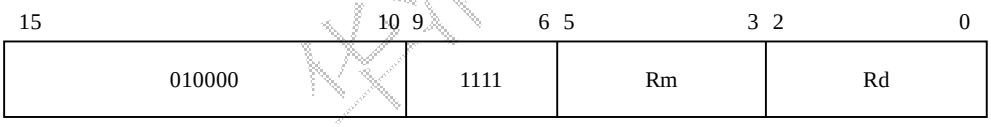


图 11.32 MVN 指令的编码格式

MVN 指令将一个数的反码传送到目标寄存器，并根据指令的执行结果更新 CPSR 中相应的条件标志位。

（2）指令的语法格式

```
MVN  <Rd>，<Rm>
```

① <Rd>

目的寄存器，存放指令的操作结果。

② <Rm>

源寄存器，存放要传送反码的数据。

（3）指令操作的伪代码

```
Rd = NOT Rm
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
```

```
C Flag = unaffected
V Flag = unaffected
```

（4）对应的 ARM 指令

```
MVNS  <Rd>，<Rm>
```

11.4.27　取反指令 NEG

（1）编码格式

取反指令 NEG 的编码格式如图 11.33 所示。

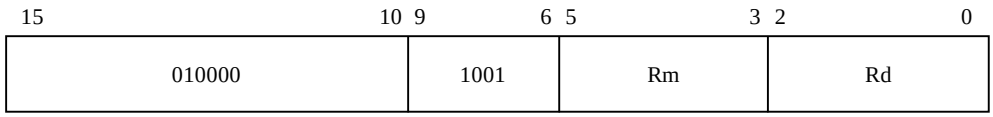


图 11.33　NEG 指令的编码格式

NEG 取反指令将一个寄存器的值取反存入另一个寄存器。同时根据指令的操作结果更新 CPSR 中相应的条件标志位。

（2）指令的语法格式

```
NEG  <Rd>，<Rm>
```

① <Rd>

目的寄存器，存放指令的操作结果。

② <Rm>

源寄存器，包含将要被取反的数据。

（3）指令操作的伪代码

```
Rd = 0 – Rm
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = NOT BorrowFrom(0 - Rm)
V Flag = OverflowFrom(0 - Rm)
```

（4）对应的 ARM 指令

```
RSBS  <Rd>，<Rm>，＃0
```

11.4.28　逻辑或指令 ORR

（1）编码格式

逻辑或指令 ORR 的编码格式如图 11.34 所示。

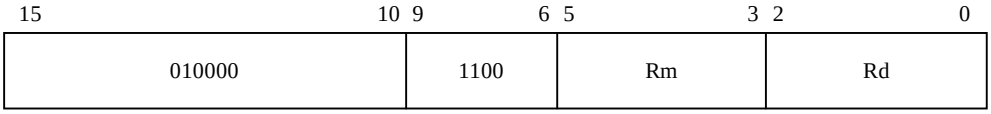


图 11.34 ORR 指令的编码格式

ORR 指令将两个寄存器的值做按位做或运算，并根据指令的执行结果更新程序状态寄存器的标志位。

（2）指令的语法格式

```
ORR  <Rd>，<Rm>
```

① <Rd>

目的寄存器，存放指令的操作结果，存放做逻辑或运算的其中一个操作数。

② <Rm>

存放做逻辑或运算的另一个操作数。

（3）指令操作的伪代码

```
Rd = Rd OR Rm
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

（4）对应的 ARM 指令

```
ORRS  <Rd>，<Rd>，<Rm>
```

11.4.29 循环右移指令 ROR

（1）编码格式

循环右移指令 ROR 的编码格式如图 11.35 所示。

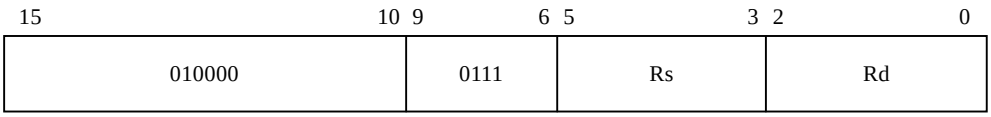


图 11.35 ROR 指令的编码格式

ROR 指令将一个给定寄存器的值循环右移一定的位数。并根据指令的操作结果更新状态寄存器的条件标志位。详见指令操作伪代码。

（2）指令的语法格式

```
ROR  <Rd>，<Rs>
```

① <Rd>

目的寄存器。存储指令的操作结果和被移位的数值。

② <Rs>

循环左移的位数。

（3）指令操作的伪代码

```
if  Rs[7:0] = = 0 then
    C Flag = unaffected
    Rd = unaffected
Else if Rs[4:0] = = 0 then
    C Flag = Rd[31]
    Rd = unaffected
Else /*Rs[4:0] > 0 */
    C Flag = Rd[Rs[4:0]- 1 ]
    Rd = Rd Rotate_Right Rs[4:0]
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
V Flag = unaffected
```

（4）对应的 ARM 指令

```
MOVS  <Rd>，<Rd>，ROR<Rs>
```

11.4.30　带进位的减指令 SBC

（1）编码格式

带进位的减指令 SBC 的编码格式如图 11.36 所示。

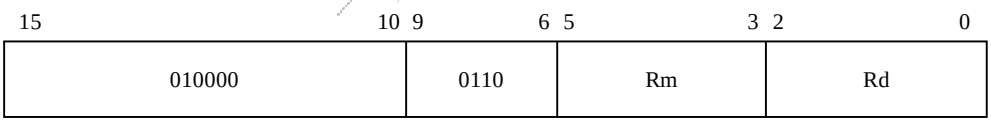


图 11.36　SBC 指令的编码格式

SBC 指令从指定寄存器中减去另一个寄存器的数值，再减去寄存器 CPSR 中 C 条件标志位的反码，并把结果保存到目标寄存器中，同时根据操作的结果更新 CPSR 中相应的条件标志位。

（2）指令的语法格式

```
SBC  <Rd>，<Rm>
```

① <Rd>

被减数寄存器，同时保存指令的操作结果。

② <Rm>

减数寄存器，保存减法操作的减数。

（3）指令操作的伪代码

```
Rd = Rd – Rm – NOT(C Flag)
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = NOT BorrowFrom(Rd – Rm – NOT(C Flag))
V Flag = Overflow From(Rd – Rm – NOT(C Flag))
```

（4）对应的 ARM 指令

```
SBCS  <Rd>，<Rd>，<Rm>
```

11.4.31 减法指令 SUB（1）

（1）编码格式

减法指令 SUB（1）的编码格式如图 11.37 所示。

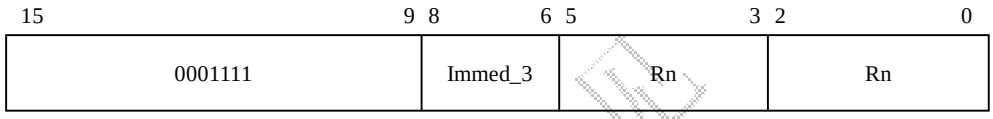


图 11.37 SUB（1）指令的编码格式

SUB（1）指令从指定寄存器减去 3 位立即数（取值范围为 0～8），并把指令的操作结果保存到寄存器，同时根据结果更新 CPSR 中相应的条件标志位。

（2）指令的语法格式

```
SUB  <Rd>，<Rn>， #<immed_3>
```

① <Rd>

目的寄存器，存放指令的操作结果。

② <Rn>

被减数寄存器，包含减法操作的被减数。

③ <immed_3>

作为减数的立即数，该立即数的取值范围为 0～8。

（3）指令操作的伪代码

```
Rd = Rn – immed_3
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = NOT BorrowFrom (Rn – immed_3)
V Flag = OverflowFrom(Rn – immed_3)
```

（4）对应的 ARM 指令

```
SUBS  <Rd>，<Rn>， #<immed_3>
```

11.4.32 减法指令 SUB（2）

（1）编码格式

减法指令 SUB（2）的编码格式如图 11.38 所示。

SUB（2）指令从指定寄存器减去 8 位立即数（即，取值范围为 0～256），并把指令的操作结果保存到寄存器，同时根据结果更新 CPSR 中相应的条件标志位。

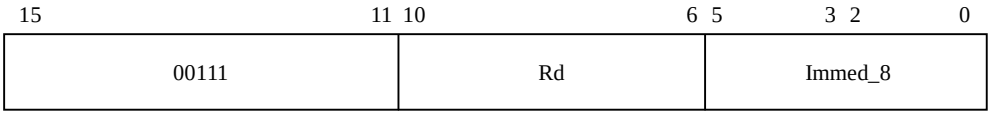


图 11.38 SUB（2）指令的编码格式

（2）指令的语法格式

```
SUB  <Rd>, #<immed_8>
```

① <Rd>

第一个源操作数所在的寄存器，存放减法操作的被减数，同时将减法操作的执行结果保存到该寄存器。

② <immed_8>

8 位立即数，即减法操作的减数，取值范围为 0～255。

（3）指令操作的伪代码

```
Rd = Rd – immed_8
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = NOT BorrowFrom (Rn – immed_8)
V Flag = OverflowFrom(Rn – immed_8)
```

（4）对应的 ARM 指令

```
SUBS  <Rd>, <Rd>, #<immed_8>
```

11.4.33 减法指令 SUB（3）

（1）编码格式

减法指令 SUB（3）的编码格式如图 11.39 所示。

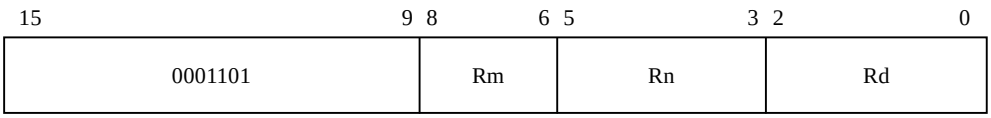


图 11.39 SUB（3）指令的编码格式

SUB（3）指令将指定寄存器的值和另一个寄存器表示的值做减法运算，并根据指令的操

作结果更新状态寄存器的标志位。

（2）指令的语法格式

```
SUB  <Rd>，<Rn>，<Rm>
```

① <Rd>

目的寄存器，存放指令操作的结果。

② <Rn>

源寄存器，存放第一个操作数，即减法运算的被减数。

③ <Rm>

源寄存器，存放第二个操作数，即减法运算的减数。

（3）指令操作的伪代码

```
Rd = Rd – Rm
N Flag = Rd[31]
Z Flag = if Rd = = 0 then 1 else 0
C Flag = NOT BorrowFrom (Rn –Rm)
V Flag = OverflowFrom(Rn – Rm)
```

（4）对应的 ARM 指令

```
SUBS  <Rd>，<Rn>，<Rm>
```

11.4.34 减法指令 SUB（4）

（1）编码格式

减法指令 SUB（4）的编码格式如图 11.40 所示。

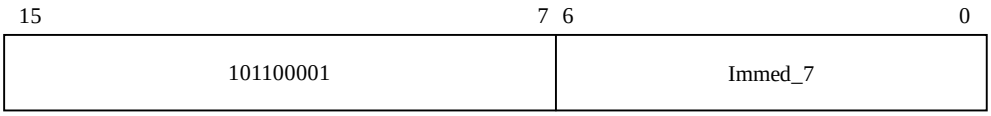


图 11.40 SUB（4）指令的编码格式

SUB（4）指令从堆栈指针 SP 中减去 7 位立即数的 4 倍，也就是说其取值为在 0～508 范围内 4 的倍数。

（2）指令的语法格式

```
SUB  SP，＃<immed_7>×4
```

① SP

程序的堆栈指针，同时也为指令的目的寄存器，存放指令的运算结果。

② <immed_7>

7 位立即数，其值的 4 倍将作为减数参加运算。

（3）指令操作的伪代码

```
SP = SP – (immed_7 << 2)
```

注意　在 Thumb 指令集中，使用满递减堆栈，该指令常被用于元素的入栈操作。

（4）对应的 ARM 指令

```
SUB  SP，SP，＃<immed_7>*4
```

11.4.35　位测试指令 TST

（1）编码格式

位测试指令 TST 的编码格式如图 11.41 所示。

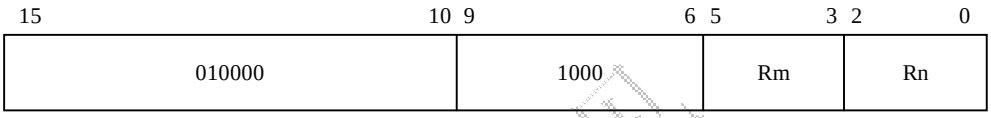


图 11.41　TST 指令的编码格式

TST 指令将两个寄存器的值按位做逻辑与操作，并根据指令的执行结果更新 CPSR 中相应的条件标志位。TST 指令常被用于测试寄存器中某一位是否置位。

（2）指令的语法格式

```
TST  <Rn>，<Rm>
```

① <Rn>

操作数寄存器，用于存放指令的第一个操作数。

② <Rm>

操作寄存器，该寄存器中的值将和<Rn>寄存器中的值做逻辑与操作。

（3）指令操作的伪代码

```
alu_out = Rn AND Rm
N Flag = alu_out[31]
Z Flag = if alu_out = = 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

（4）对应的 ARM 指令

```
TST  <Rn>，<Rm>
```

11.4.36　Thumb 指令集中数据操作指令举例

下面的例子程序综合使用了各种数据操作指令，通过该例可以对 Thumb 状态下数据操作

指令有更深入的了解。

```
ADD  r0,r4,r7      ;r0 = r4 + r7
SUB  r6,r1,r2      ;r6 = r1 – r2
ADD  r0,#255       ;r0 = r0 + 255
ADD  r1,r4,#4      ;r1 = r4 + 4
NEG  r3,r1         ;r3 = 0 – r1
ADD  r2,r5         ;r2 = r2 AND r5
EOR  r1,r6         ;r1 = r1 EOR r6
CMP  r2,r3         ;r2 – r3 ，并更新 CPSR
CMP  r7,#100       ;r7 – 100 ，并更新 CPSR
MOV  r0,#200       ;r0 = 200
```

11.5 单寄存器数据传送指令

Thumb 指令集支持寄存器的装载和存储，即 LDR 和 STR 指令。8 和类型的 Load/Store 指令在 Thumb 指令集中可用。这些指令使用两种寻址模式：寄存器偏移和立即数偏移。指令所能存取的数据包括字、半字和字节，同时半字和字节可以为有符号数或无符号数。

表 11.4 总结了 Thumb 状态下可用的数据传送指令。

表 11.4 Thumb 状态数据传送指令

助 记 符	说 明	操 作
LDR	传送 32 位字到寄存器	Rd<— mem32[address]
STR	存储 32 位寄存器的值	Rd—> mem32[address]
LDRB	传送 8 位字节到寄存器	Rd<— mem8[address]
STRB	保存寄存器中的字节	Rd—> mem8[address]
LDRH	传送 16 位半字到寄存器	Rd<— mem16[address]
STRH	保存寄存器中的半字	Rd—> mem16[address]
LDRSB	装载有符号字节到寄存器	Rd<— sighExtend(mem8[address])
STRSB	装载有符号半字到寄存器	Rd<— sighExtend(mem16[address])

Thumb 数据传送指令的基本语法格式分为以下 4 种。

① <opcode1> <Rd>，[<Rn>，#<5_bit_offset>]

其中，<opcode1>：= LDR|LDRH|LDRB|STR|STRH|STRB

② <opcode2> <Rd>，[<Rn>，<Rm>]

其中，<opcode2>：= LDR|LDRH|LDRB|LSRSH|STR|STRH|STRB

```
③ LDR  <Rd>，[PC，<8_bit_offset>]
④ <opcode3>  <Rd>，[SP，＃<8_bit_offset>]
```

其中，<opcode3>：= LDR|STR

下面详细介绍各数据传送指令的语法和使用。

11.5.1 寄存器装载指令 LDR（1）

（1）编码格式

寄存器装载指令 LDR（1）的编码格式如图 11.42 所示。

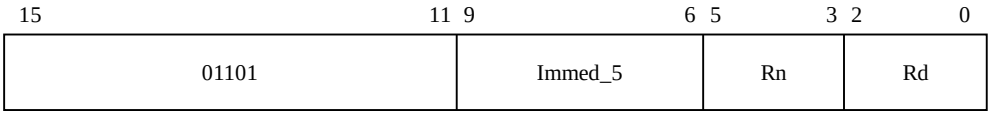


图 11.42　LDR（1）指令的编码格式

这种形式的 LDR 指令将 32 位内存数据装载到通用寄存器。常用于结构体的数据访问。域的基地址放在 Rn 寄存器中。

（2）指令的语法格式

```
LDR  <Rd>，[<Rn>，＃<immed_5>*4]
```

① <Rd>

目的寄存器。用于存放从内存中取出的数据。

② <Rn>

基址寄存器，用于存放所取数据的基地址。

③ <immed_5>

5 位立即数。该立即数的 4 倍加上基址寄存器的值形成目标地址。

（3）指令操作的伪代码

```
Address = Rn + (immed_5 * 4)
If  address[1:0] = = 0b00
    Data = Memory[address,4]
Else
    Data = UNPREDICTABLE
Rd = data
```

（4）对应的 ARM 指令

```
LDR  <Rd>，[<Rn>，＃<immed_5>*4]
```

注意　如果指令访问地址非字对齐，则指令的执行结果不可预知。

11.5.2 寄存器装载指令 LDR（2）

（1）编码格式

寄存器装载指令 LDR（2）的编码格式如图 11.43 所示。

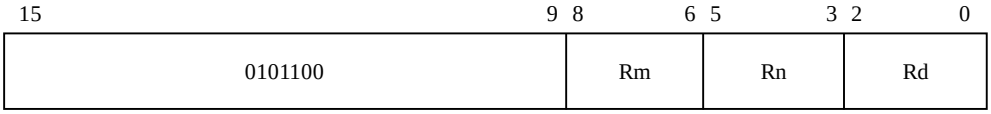


图 11.43 LDR（2）指令的编码格式

寄存器装载指令 LDR（2）允许将一个 32 位内存数据装载到通用寄存器。此种形式的 LDR 指令常被用于访问数组中的元素。

（2）指令的语法格式

```
LDR  <Rd>，[<Rn>，<Rm>]
```

① <Rd>

目的寄存器。

② <Rn>

寄存器存放内存访问基地址。

③ <Rm>

寄存器存放内存访问偏移地址。

（3）指令操作的伪代码

```
Address = Rn + Rm
If  address[1:0] = = 0b00
    Data = Memory[address,4]
Else
    Data = UNPREDICTABLE
Rd = data
```

（4）对应的 ARM 指令

```
LDR  <Rd>，[<Rn>，<Rm>]
```

11.5.3 寄存器装载指令 LDR（3）

（1）编码格式

寄存器装载指令 LDR（3）的编码格式如图 11.44 所示。

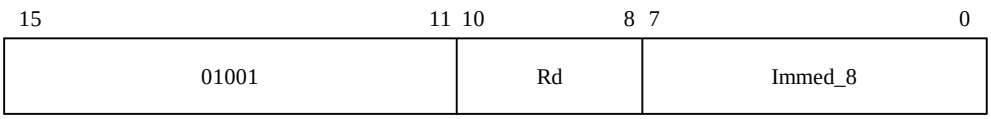


图 11.44 LDR（3）指令的编码格式

寄存器装载指令 LDR（3）允许将一个 32 位内存数据装载到通用寄存器。此种形式的 LDR 指令常被用于访问 PC 相关（PC-relative）数据。

（2）指令的语法格式

```
LDR  <Rd>，[PC， #<immed_8>*4]
```

① <Rd>

目的寄存器。

② PC

程序指针寄存器，用于计算内存访问的地址。计算地址时，PC 值的 bit[1]被系统默认为 0 进行计算，所以产生的内存访问地址必为字对齐。

③ <immed_8>

8 位立即数。该立即数的 4 倍将和 PC 值相加，形成内存访问地址。

（3）指令操作的伪代码

```
Address = (PC[31:2] << 2) + (immed_8*4)
Rd = Memory[address,4]
```

（4）对应的 ARM 指令

```
LDR  <Rd>，[PC， #<immed_8>*4]
```

11.5.4 寄存器装载指令 LDR（4）

（1）编码格式

寄存器装载指令 LDR（4）的编码格式如图 11.45 所示。

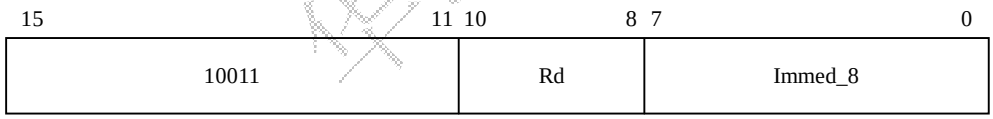


图 11.45 LDR（4）指令的编码格式

寄存器装载指令 LDR（4）允许将一个 32 位内存数据装载到通用寄存器。此种形式的 LDR 指令常被用于访问堆栈数据。

（2）指令的语法格式

```
LDR  <Rd>，SP， #<immed_8>*4]
```

① <Rd>

目的寄存器。

② SP

堆栈指针寄存器，用于计算内存访问地址。

③ <immed_8>

8 位立即数。该立即数的 4 倍将和 SP 值相加，形成内存访问地址。

（3）指令操作的伪代码

```
Address = SP + (immed_8*4)
If  address[1:0] = = 0b00
    Data = memory[address,4]
Else
    Data = UNPREDICTABLE
Rd = data
```

（4）对应的 ARM 指令

```
LDR  <Rd>，[SP， #<immed_8>*4]
```

11.5.5 字节加载指令 LDRB（1）

（1）编码格式

字节加载指令 LDRB（1）的编码格式如图 11.46 所示。

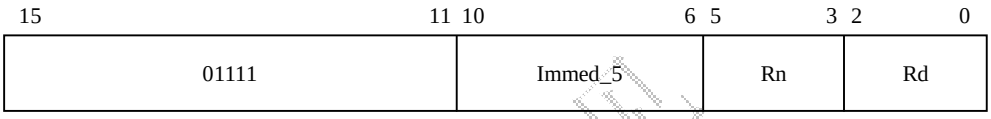


图 11.46 LDRB（1）指令的编码格式

LDRB（1）字节数据加载指令用于从内存中将一个 8 位的字节数据读取到指令中的目标寄存器中，并将寄存器的高 24 位清零。常用于结构体的数据访问。域的基地址放在 Rn 寄存器中。

（2）指令的语法格式

```
LDRB  <Rd>，[<Rn>， #<immed_5>]
```

① <Rd>

目的寄存器。

② <Rn>

指令的基址寄存器。

③ <immed_5>

5 位立即数。用于与<Rn>寄存器中的数值相加，形成内存访问地址。

（3）指令操作的伪代码

```
address = Rn + immed_5
Rd = memory[address,1]
```

（4）对应的 ARM 指令

```
LDRB  <Rd>，[<Rn>， #<immed_5>]
```

11.5.6 字节加载指令 LDRB（2）

（1）编码格式

字节加载指令 LDRB（2）的编码格式如图 11.47 所示。

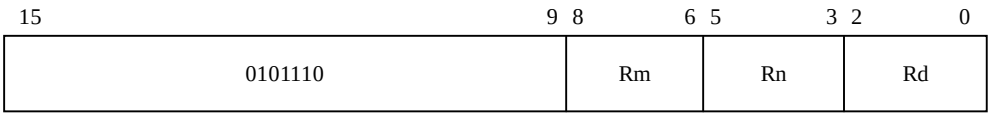


图 11.47　LDRB（2）指令的编码格式

LDRB（2）字节数据加载指令用于从内存中将一个 8 位的字节数据读取到指令中的目标寄存器中，并将寄存器的高 24 位清零。此种形式的 LDRB（2）指令常用于数组元素的访问。

（2）指令的语法格式

```
LDRB  <Rd>，[<Rn>，<Rm>]
```

① <Rd>

目的寄存器。

② <Rn>

存放形成内存访问地址的第一个寄存器。

③ <Rm>

存放形成内存访问地址的第二个寄存器。

（3）指令操作的伪代码

```
address = Rn + Rm
Rd = Memory[address,1]
```

（4）对应的 ARM 指令

```
LDRB  <Rd>，[<Rn>，<Rm>]
```

11.5.7　半字加载指令 LDRH（1）

（1）编码格式

半字数据加载指令 LDRH（1）的编码格式如图 11.48 所示。

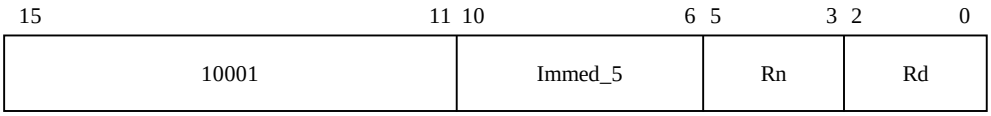


图 11.48　LDRH（1）指令的编码格式

LDRH（1）半字数据加载指令用于从内存中将一个 16 位的半字数据读取到指令中的目标寄存器中，并将寄存器的高 16 位清零。常用于结构体的数据访问。域的基地址放在 Rn 寄存器中。

（2）指令的语法格式

```
LDRH  <Rd>，[<Rn>，＃<immed_5>*2]
```

① <Rd>

目的寄存器。

② <Rn>

指令的基址寄存器。

③ <immed_5>

5 位立即数。该寄存器数值的 2 倍将与<Rn>寄存器中的数值相加，形成内存访问地址。

（3）指令操作的伪代码

```
address = Rn + (immed_5 *2)
if  address[0] = = 0
    data = Memory[address,2]
else
    data = UNPREDICTABLE
Rd = data
```

（4）对应的 ARM 指令

```
LDRH  <Rd>，[<Rn>， #<immed_5>*2]
```

11.5.8 半字数据加载指令 LDRH（2）

（1）编码格式

半字数据加载指令 LDRH（2）的编码格式如图 11.49 所示。

LDRH（2）字节数据加载指令用于从内存中将一个 16 位的半字数据读取到指令中的目标寄存器中，并将寄存器的高 16 位清零。此种形式的 LDRH（2）指令常用于数组元素的访问。

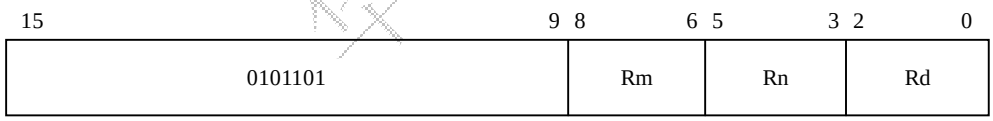


图 11.49 LDRH（2）指令的编码格式

（2）指令的语法格式

```
LDRB  <Rd>，[<Rn>，<Rm>]
```

① <Rd>

目的寄存器。

② <Rn>

此寄存器存放内存访问基地址。

③ <Rm>

此寄存器存放内存访问偏移地址。

（3）指令操作的伪代码

```
address = Rn + Rm
```

```
if  address[0] = = 0
    data = memory[address,2]
else
    data = UNPREDICTABLE
Rd = data
```

（4）对应的 ARM 指令

```
LDRH  <Rd>，[<Rn>，<Rm>]
```

11.5.9　有符号字节数据加载指令 LDRSB

（1）编码格式

有符号字节数据加载指令 LDRSB 的编码格式如图 11.50 所示。

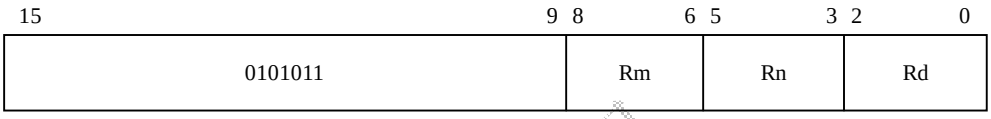


图 11.50　LDRSB 指令的编码格式

LDRSB 指令用于从内存中将一个 8 位的字节数据读取到指令中的目标寄存器中，并将寄存器的高 24 位设置成该字节数据的符号位的值（即将该 8 位字节数据进行符号位扩展，生成 32 位字数据）。

（2）指令的语法格式

```
LDRSB  <Rd>，[<Rn>，<Rm>]
```

① <Rd>

目的寄存器。

② <Rn>

此寄存器存放内存访问基地址。

③ <Rm>

此寄存器存放内存访问偏移地址。

（3）指令操作的伪代码

```
address = Rn + Rm
Rd = SignExtend(Memory[address,1])
```

（4）对应的 ARM 指令

```
LDRSB  <Rd>，[<Rn>，<Rm>]
```

11.5.10　有符号半字数据加载指令 LDRSH

（1）编码格式

有符号字节数据加载指令 LDRSH 的编码格式如图 11.51 所示。

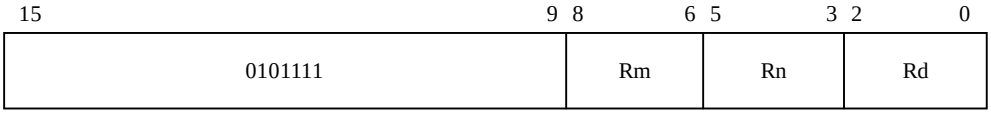


图 11.51 LDRSH 指令的编码格式

LDRSH 指令用于从内存中将一个 16 位的半字数据读取到指令中的目标寄存器中，并将寄存器的高 16 位设置成该半字数据的符号位的值（即将该 16 位半字数据进行符号位扩展，生成 32 位字数据）。

（2）指令的语法格式

```
LDRBH  <Rd>，[<Rn>，<Rm>]
```

① <Rd>

目的寄存器。

② <Rn>

此寄存器存放内存访问基地址。

③ <Rm>

此寄存器存放内存访问偏移地址。

（3）指令操作的伪代码

```
address = Rn + Rm
if  address[0] = = 0
    data = memory[address,2]
else
    data = UNPREDICTABLE
Rd = SignExtend[data]
```

（4）对应的 ARM 指令

```
LDRSH  <Rd>，[<Rn>，<Rm>]
```

11.5.11 寄存器存储指令 STR（1）

（1）编码格式

寄存器存储指令 STR（1）的编码格式如图 11.52 所示。

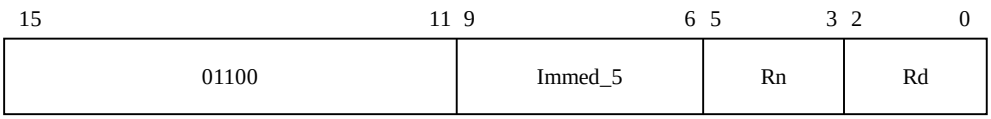


图 11.52 STR（1）指令的编码格式

这种形式的 STR 指令将 32 位通用寄存器的数值存储到内存中。该指令常用于结构体的数据访问。域的基地址放在 Rn 寄存器中。

（2）指令的语法格式

```
STR  <Rd>，[<Rn>， #<immed_5>*4]
```

① <Rd>

目的寄存器。用于存放从内存中取出的数据。

② <Rn>

基址寄存器，用于存放所取数据的基地址。

③ <immed_5>

5 位立即数。该立即数的 4 倍加上基址寄存器的值为目标地址。

（3）指令操作的伪代码

```
address = Rn + (immed_5*4)
if  address[1:0] = = 0b00
     Memory[address,4] = Rd
Else
     Memory[address,4] = UNPREDICTABLE
```

（4）对应的 ARM 指令

```
STR  <Rd>，[<Rn>， #<immed_5>*4]
```

11.5.12 寄存器存储指令 STR（2）

（1）编码格式

寄存器存储指令 STR（2）的编码格式如图 11.53 所示。

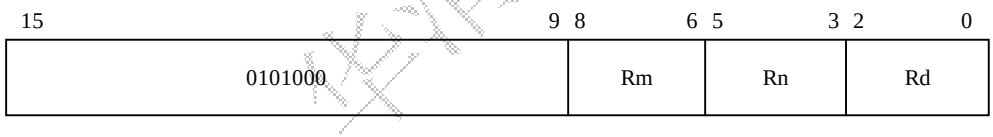


图 11.53 STR（2）指令的编码格式

寄存器装载指令 STR（2）将一个 32 位通用寄存器数据存储到内存单元中。此种形式的 STR 指令常被用于访问数组中的元素。

（2）指令的语法格式

```
LDR  <Rd>，[<Rn>，<Rm>]
```

① <Rd>

目的寄存器。

② <Rn>

存放形成内存访问地址的第一个寄存器。

③ <Rm>

存放形成内存访问地址的第二个寄存器。

（3）指令操作的伪代码

```
address = Rn + Rm
```

```
if  address[1:0] = = 0b00
     Memory[address,4] = = Rd
Else
     Memory[address,4] = = UNPREDICTABLE
```

（4）对应的 ARM 指令

```
STR  <Rd>，[<Rn>，<Rm>]
```

11.5.13 寄存器存储指令 STR（3）

（1）编码格式

寄存器存储指令 STR（3）的编码格式如图 11.54 所示。

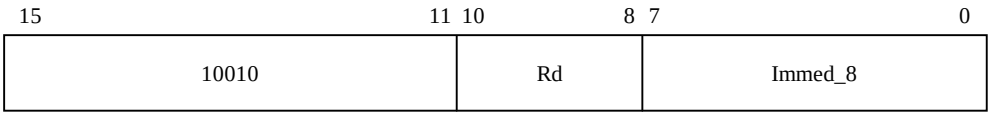


图 11.54 STR（3）指令的编码格式

寄存器存储指令 STR（3）允许将一个 32 位通用寄存器的值存储到内存。此种形式的 STR 指令常被用于访问堆栈数据。

（2）指令的语法格式

```
STR  <Rd>，[SP， #<immed_8>*4]
```

① <Rd>

目的寄存器。

② SP

堆栈指针寄存器，用于计算内存访问的地址。

③ <immed_8>

8 位立即数。该立即数的 4 倍将和堆栈指针寄存器 SP 的值相加，形成内存访问地址。

（3）指令操作的伪代码

```
address = SP + (immed_8 * 4)
if  address[1:0] = = 0b00
     Memory[address,4] = Rd
Else
     Memory[address,4] = UNPREDICTABLE
```

（4）对应的 ARM 指令

```
STR  <Rd>，[SP， #<immed_8>*4]
```

11.5.14 字节存储指令 STRB（1）

（1）编码格式

字节存储加载指令 STRB（1）的编码格式如图 11.55 所示。

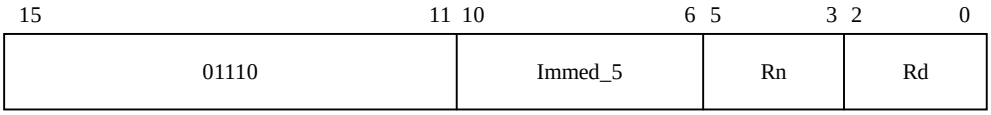


图 11.55　STRB（1）指令的编码格式

STRB（1）字节数据存储指令用于将一个 8 位的字节数据写入到指令中指定的内存单元，该字节数据为指令中存放源操作数寄存器的低 8 位。常用于结构体的数据访问。域的基地址放在 Rn 寄存器中。

（2）指令的语法格式

```
STRB  <Rd>，[<Rn>， #<immed_5>]
```

① <Rd>

目的寄存器。

② <Rn>

指令的基址寄存器。

③ <immed_5>

5 位立即数。用于与<Rn>寄存器中的数值相加，形成内存访问地址。

（3）指令操作的伪代码

```
address = Rn + immed_5
Memory[address,1] = Rd[7:0]
```

（4）对应的 ARM 指令

```
STRB  <Rd>，[<Rn>， #<immed_5>]
```

11.5.15　寄存器存储指令 STRB（2）

（1）编码格式

寄存器存储指令 STRB（2）的编码格式如图 11.56 所示。

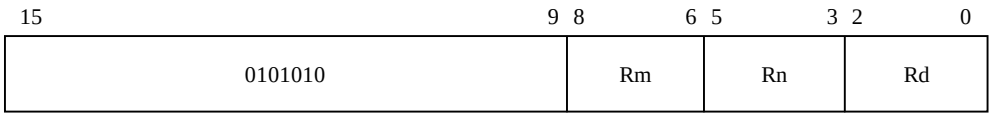


图 11.56　STRB（2）指令的编码格式

寄存器存储指令 STRB（2）用于将一个 8 位的字节数据写入到指令中指定的内存单元。此种形式的 LDRB 指令常被用于访问数组中的元素。

（2）指令的语法格式

```
STRB  <Rd>，[<Rn>，<Rm>]
```

① <Rd>

目的寄存器。

② <Rn>

此寄存器存放内存访问基地址。

③ <Rm>

此寄存器存放内存访问偏移地址。

（3）指令操作的伪代码

```
address = Rn + Rm
Memory[address,1] = Rd[7:0]
```

（4）对应的 ARM 指令

```
STRB  <Rd>，[<Rn>，<Rm>]
```

11.5.16 半字存储指令 STRH（1）

（1）编码格式

半字存储加载指令 STRH（1）的编码格式如图 11.57 所示。

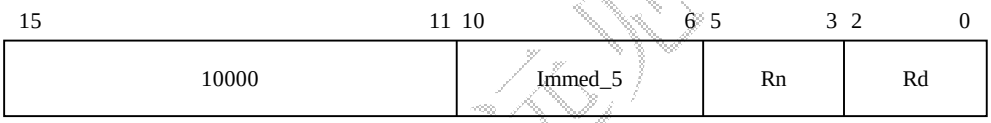


图 11.57 STRH（1）指令的编码格式

STRH（1）半字数据存储指令用于将一个 16 位的半字数据写入到指令中指定的内存单元，该半字数据为指令中存放源操作数寄存器的低 16 位。常用于结构体的数据访问。域的基地址放在 Rn 寄存器中。

（2）指令的语法格式

```
STRH  <Rd>，[<Rn>，＃<immed_5>*2]
```

① <Rd>

目的寄存器。

② <Rn>

指令的基址寄存器。

③ <immed_5>

5 位立即数。该立即数的 2 倍与<Rn>寄存器中的数值相加，形成内存访问地址。

（3）指令操作的伪代码

```
address = Rn + (immed_5*2)
if  address[1:0] = = 0
     Memory[address,2] = Rd[15:0]
Else
     Memory[address,2] = UNPREDICTABLE
```

（4）对应的 ARM 指令

```
STRH  <Rd>，[<Rn>， #<immed_5>*2]
```

11.5.17　寄存器存储指令 STRH（2）

（1）编码格式

寄存器存储指令 STRH（2）的编码格式如图 11.58 所示。

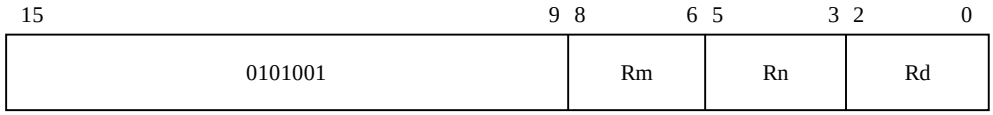


图 11.58　STRH（2）指令的编码格式

寄存器存储指令 STRH（2）用于将一个 8 位的半字数据写入到指令中指定的内存单元。此种形式的 STRH 指令常被用于访问数组中的元素。

（2）指令的语法格式

```
STRH  <Rd>，[<Rn>，<Rm>]
```

① <Rd>

目的寄存器。

② <Rn>

存放形成内存访问地址的第一个寄存器。

③ <Rm>

存放形成内存访问地址的第二个寄存器。

（3）指令操作的伪代码

```
address = Rn + Rm
if  address[1:0] = = 0
     Memory[address,2] = Rd[15:0]
Else
     Memory[address,2] = UNPREDICTABLE
```

（4）对应的 ARM 指令

```
STRH  <Rd>，[<Rn>，<Rm>]
```

11.5.18　数据传送指令举例

下面的例子程序综合使用了各种数据传送指令，通过该例可以对 Thumb 状态下数据传送指令有更深入的了解。

```
LDR  r4，[r2，＃4]          ;将[r2+4]地址单元字数据加载到寄存器 r4
LDR  r4，[r2，r1]           ;将[r2+r4]地址单元字数据加载到寄存器 r4
```

```
STR  r0，[r7，＃0x7c]      ;将 r0 中的字数据存储到[r7+124]的内存地址单元中
STRB  r1，[r5，＃31]       ;将 r1 的低 8 位数据存储到[r5+31]的内存地址单元中
STRH  r4，[r2，r3]         ;将 r4 的低 16 位数据存储到[r2+r3]的内存地址单元中
LDRH  r3，[r6，r5]         ;将[r6+r5]地址单元低 16 位数据加载到寄存器 r3 中
LDRB  r2，[r1，＃5]        ;将[r1+5]地址单元低 8 位数据加载到寄存器 r2 中
LDR  r6，[PC，＃0xFC]      ;将[PC+0x3FC]地址单元数据加载到寄存器 r6 中
LDR  r5，[SP，＃64]        ;将[SP+64]地址单元数据加载到寄存器 r5 中
STR  r4，[SP，＃0x260]     ;将寄存器 r4 中的数据存储到[SP+0x260]内存地址单元中
```

11.6 多寄存器数据传送指令

Thumb 指令集的多寄存器 Load/Store 指令是 ARM 指令集的多寄存器 Load/Store 指令的简化形式。同 ARM 指令一样，Thumb 多寄存器数据传送指令可以用于过程调用与返回以及存储器块拷贝。但为了编码的紧凑性，这两种用法由分开的指令实现，并且这些指令也只使用单一的寻址方式。在其他方面，这些指令的性质与等价的 ARM 指令相同。

Thumb 多寄存器数据传送指令的基本语法格式分为以下两种，一种用于实现块拷贝，另一种用于实现堆栈操作。

```
① <opcode1>  <Rn>!，<registers>
<opcode1>: =LDMIA|STMIA
② PUSH  {<registers>}
POP  {<registers>}
```

下面详细介绍多寄存器数据传送指令的语法格式及用法。

11.6.1 多寄存器数据加载指令 LDMIA

（1）编码格式

多寄存器加载指令 LDMIA 的编码格式如图 11.59 所示。

多寄存器数据装载指令 LDMIA（Load Multiple Increment After）装载连续的内存单元到多个通用寄存器。

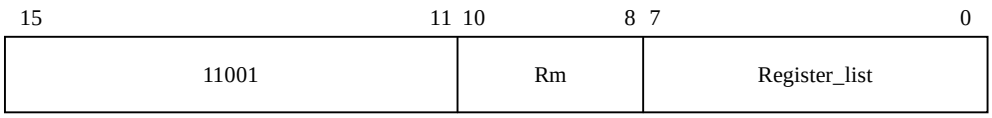


图 11.59 LDMIA 指令的编码格式

（2）指令的语法格式

```
LDMIA  <Rn>!，<registers>
```

① <Rn>

基址寄存器。指定被装载的内存单元基地址。

② ！

采用回写（writeback）的寻址方式。Thumb 指令集中多寄存器数据传送指令采用固定的后增量 IA（Increment After）寻址方式，并且采用寄存器回写方式。

③ <registers>

被加载的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中高地址单元。寄存器 r0～r7 分别对应于指令编码中 bit[0]～bit[7]位。如果 Ri 存在于寄存器列表中，则相应的 bit[i]指 1，否则为 0。

注意 如果基址寄存器 Rn 出现在寄存器列表<registers>中，则 Rn 的值为对于的内存单元数据，而非地址回写值。

（3）指令操作的伪代码

```
Start_address = Rn
End_address = Rn + (Number_of_Set_Bits_In(register_list)*4) – 4
Address = start_address
For i = 0 to 7
     If  register_list[i] = = 1
        Ri = Memory[address,4]
        Address = address + 4
Assert end_address = = address – 4
Rn = Rn + (Number_of_Set_Bits_In(register_list) * 4)
```

（4）对应的 ARM 指令

① 如果基址寄存器在寄存器列表<registers>中，对应的 ARM 指令为：

```
LDMIA  <Rn>!，<registers>。
```

② 如果基址寄存器不存在于寄存器列表<registers>中，对应的 ARM 指令为：

```
LDMIA  <Rn>，<registers>
```

11.6.2 多寄存器数据存储指令 STMIA

（1）编码格式

多寄存器数据存储指令 STMIA 的编码格式如图 11.60 所示。

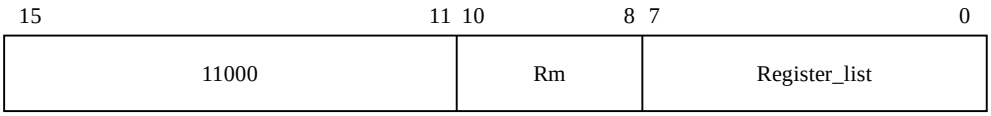


图 11.60　STMIA 指令的编码格式

多寄存器数据存储指令 STMIA（Store Multiple Increment After）存储多个通用寄存器的

内容到连续的内存单元。

（2）指令的语法格式

```
STMIA  <Rn>!，<registers>
```

① <Rn>

基址寄存器。指定用于存储的内存单元基地址。

② !

采用回写（writeback）的寻址方式。Thumb 指令集中多寄存器数据传送指令采用固定的后增量 IA（Increment After）寻址方式，并且采用寄存器回写方式。

③ <registers>

被存储的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中高地址单元。寄存器 r0～r7 分别对应于指令编码中 bit[0]～bit[7]位。如果 Ri 存在于寄存器列表中，则相应的 bit[i]置 1，否则为 0。如果寄存器列表为空，即指令的编码格式中 bit[7：0] = 0，则指令的执行结果不可预知。

注意 当基址寄存器 Rn 出现在寄存器列表<registers>中时，只能是列表中序号最低的寄存器，否则指令的执行结果不可预知。

（3）指令操作的伪代码

```
Start_address = Rn
End_address = Rn + (Number_of_Set_Bits_In(register_list)*4) – 4
Address = start_address
For i = 0 to 7
     If  register_list[i] = = 1
        Memory[address,4] = Ri
        Address = address + 4
Assert end_address = = address – 4
Rn = Rn + (Number_of_Set_Bits_In(register_list) * 4)
```

（4）对应的 ARM 指令

```
STMIA  <Rn>!，<registers>
```

11.6.3 多寄存器压栈指令 PUSH

（1）编码格式

多寄存器压栈指令 PUSH 的编码格式如图 11.61 所示。

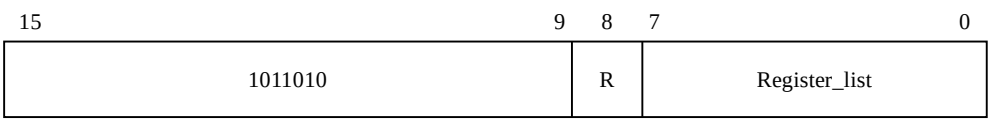


图 11.61 PUSH 指令的编码格式

多寄存器数据压栈指令 PUSH（Push Multiple Registers）将 r0～r7 和 LR 中的一个或多个寄存器内容加载到数据堆栈中。

（2）指令的语法格式

```
PUSH  <registers>
```

<registers>

被存储的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。寄存器 r0～r7 分别对应于指令编码中 bit[0]～bit[7]位；返回连接寄存器 LR 对应于 bit[8]。如果 ri 存在于寄存器列表中，则相应的 bit[i]置 1，否则为 0。如果寄存器列表为空，即 bit[8：0] = 0，则指令的执行结果不可预知。

注意 该指令的基址寄存器为堆栈寄存器 SP。该基址寄存器为 PUSH 指令默认寄存器，不必在指令中指定。

（3）指令操作的伪代码

```
Start_address = SP – Number_of_Set_Bits_In(register_list)*4
End_address = SP – 4
Address = start_address
For i = 0 to 7
     If  register_list[i] = = 1
        Memory[address,4] = Ri
        Address = address + 4
If  R = = 1
    Memory[address,4] = LR
    Address = address + 4
Assert end_address = = address – 4
SP = SP - (Number_of_Set_Bits_In(register_list) + R) * 4
```

（4）对应的 ARM 指令

```
STMDB  SP!，<registers>
```

11.6.4　多寄存器出栈指令 POP

（1）编码格式

多寄存器出栈指令 POP 的编码格式如图 11.62 所示。

多寄存器数据出栈指令 POP（Pop Multiple Registers）将堆栈中的内容恢复到 r0～r7 和 PC 寄存器中（r0～r7 和 PC 的子集或全集）。

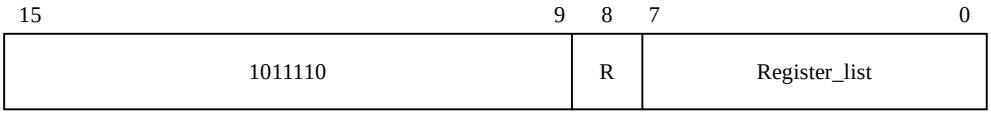


图 11.62 POS 指令的编码格式

（2）指令的语法格式

```
POP  <registers>
```

① <registers>

被存储的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。寄存器 r0～r7 分别对应于指令编码中 bit[0]～bit[7]位；程序计数器 PC 对应于 bit[8]。如果 ri 存在于寄存器列表中，则相应的 bit[i]置 1，否则为 0。如果寄存器列表为空，即 bit[8：0] =0，则指令的执行结果不可预知。

注意 如果程序计数器 PC 存在于寄存器列表中，将产生程序的分支跳转，但程序状态寄存器 CPSR 不会改变。

（3）指令操作的伪代码

```
Start_address = SP
End_address = SP + 4*(R + Number_of_Set_Bits_In(register_list))
Address = start_address

For i = 0 to 7
     If  register_list[i] = = 1  then
        Ri = Memory[address,4]
        Address = address + 4

If  R = =1 then
     value = Memory[address,4]
     PC= value AND 0xfffffffe
     If  (architecture version 5 or above )  then
        T Bit = Value[0]
        Address = address + 4

Assert end_address = address
SP = end_address
```

（4）对应的 ARM 指令

```
LDMIA  SP!，<registers>
```

11.6.5 多寄存器数据传送指令举例

下面的例子程序综合使用了多寄存器数据传送指令，通过该例可以对 Thumb 状态下多寄存器数据传送指令有更深入的了解。

```
        LDMIA  r7!，{r0－r3，r5} ;从 r7 为基地址的内存单元中加载 5 个连续字单元到 r0～r3
```

```
                                    ;和 r5，并回写 r7，使 r7 = r7 + 20
        STMIA  r0!，{r3，r4，r5} ;将 r3、r4 和 r5 寄存器的内容存储到以 r0 为基地址的内存单
                                    ;元中，并回写 r0 的值，使 r0 = r0 + 12

    Function
        PUSH  {r0-r7，LR}          ;将寄存器 r0～r7 和程序返回寄存器 LR 的内容压栈
    …
    …                               ;程序体
        POP  {r0-r7，PC}           ;保存的寄存器内容出栈，子程序返回
```

11.7 异常中断产生指令（断点指令）

Thumb 异常中断产生指令与 ARM 指令集下的异常中断指令十分相似。同 ARM 指令集相同，Thumb 指令集中同样包含两条异常中断产生指令：软件中断指令 SWI 用于产生 SWI 异常中断；断点中断指令 BKPT 主要用于产生软件断点，供调试程序使用（只在 ARMv5 及以上版本中使用）。

11.7.1 软中断指令 SWI

（1）编码格式

软中断指令 SWI 的编码格式如图 11.63 所示。

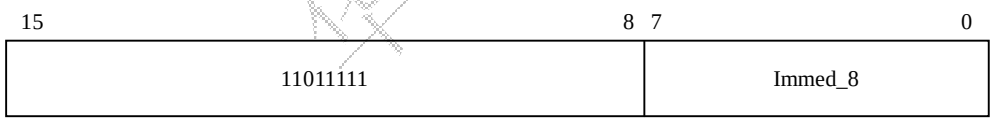


图 11.63 SWI 指令的编码格式

软中断指令 SWI（Software Interrupt）用于使处理器产生软中断异常，使用这种机制实现在用户模式对操作系统中特权模式的程序调用。

（2）指令的语法格式

```
    SWI  <immed_8>
```

其中，immed_8 为 8 位立即数，该立即数被处理器忽略，但可以被操作系统用来判断用户程序请求的服务类型。

（3）指令操作的伪代码

```
    R14_svc = address of next instruction after the SWI instruction
    SPSR_svc = CPSR
    CPSR[4:0] = 0b10011                /*进入超级模式*/
```

```
CPSR[5] = 0                          /*进入 ARM 状态*/
/*CPSR[6] is unchanged*/
CPSR[7] = 1                          /*禁止正常中断*/
If high vectors configured then
     PC = 0xffff0008
Else
     PC = 0x00000008
```

（4）对应的 ARM 指令

```
SWI  <immed_8>
```

11.7.2 断点中断指令 BKPT

（1）编码格式

断点中断指令 BKPT 的编码格式如图 11.64 所示。

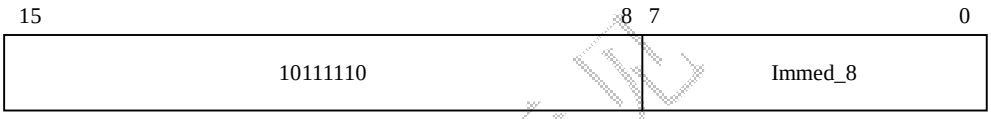


图 11.64 BKPT 指令的编码格式

断点中断指令 BKPT（Breakpoint）可以使处理器产生预取异常（Prefetch Abort），使用这种机制可以在没有调试硬件的情况下，实现程序的软件调试。当系统中使用硬件调试部件时，可忽略该中断。

（2）指令的语法格式

```
BKPT  <immed_8>
```

其中，immed_8 为 8 位立即数，该立即数被处理器忽略，但可以向调试系统提供附加的断点信息。

注意 当系统中存在调试硬件时，BKPT 指令有两种处理方法：一种由调试硬件处理 BKPT 指令；另一种可以直接由预取异常处理函数来处理 BKPT 指令，使用这种方法时，由于使用了异常模式下的链接地址寄存器 r14_abt 和程序状态寄存器 SPSR_abt，所有要特别注意程序的嵌套。

（3）指令操作的伪代码

```
if  (not overridden by debug hardware)
     R14_abt = address of BKPT instruction +4
     SPSR_abt = CPSR
     CPSR[4:0] = 0b10111          /*进入异常模式*/
     CPSR[5] = 0                  /*进入 ARM 状态*/
/*CPSR[6] is unchanged*/
     CPSR[7] = 1
```

```
        If high vectors configured then
             PC = 0xffff000c
        Else
             PC = 0x0000000c
```

（4）对应的 ARM 指令

```
BKPT  <immed_8>
```

11.8 未定义的指令空间

Thumb 指令集中存在未定义的指令空间，如图 11.65 所示。

15 14 13 12	11	10	9	8	7 … 0
1011	0	0	0	1	XXXXXXXX
1011	0	X	1	XXXXXXXXX	
1011	1	0	XXXXXXXXXX		
1011	1	1	1	1	XXXXXXXX
1101	1	1	1	0	XXXXXXXX

图 11.65 未定义的指令空间（Thumb 指令集）

这些未定义的指令空间，可以用作以后的指令集扩展。另外，用户在编写自己的应用程序时有时也要扩展自己的指令空间（如 DSP 算法的引入），这时，就可以使用这些未定义的指令空间。

11.9 Thumb 指令应用

11.9.1 Thumb 的实现

对 3 级流水线的 ARM 处理器来说，做相对较小的改动就可以实现 Thumb 指令集（5 级流水线的实现要复杂些）。为实现 Thumb 指令集，在指令流水线中增加了 Thumb 指令解码逻辑，该解码逻辑将预取的 Thumb 指令转换成等价的 ARM 指令。图 11.66 显示了 Thumb 指令的扩展逻辑组织。

从图 11.66 可以看出，Thumb 指令增加了解码扩展逻辑（Decompressor）与指令译码器串连，但这并不会增加指令的译码时间。在指令流水线译码周期的第一阶段只做了很少的工作，因此可以把扩展逻辑安排在这里而不会影响周期时间或增加流水线延时。

Thumb 指令解码扩展逻辑将 16 位的 Thumb 指令静态地转换为等价的 32 位 ARM 指令。

这主要包括主操作码和次操作码的查表转换，3 位寄存器指示符（specifier）零扩展成 4 位寄存器指示符，以及所需要的其他域的映射。

例如，Thumb 指令“ADD Rd，＃imm8”与对应的 ARM 指令“ADD Rd，Rd，＃8”的映射如图 11.67 所示。

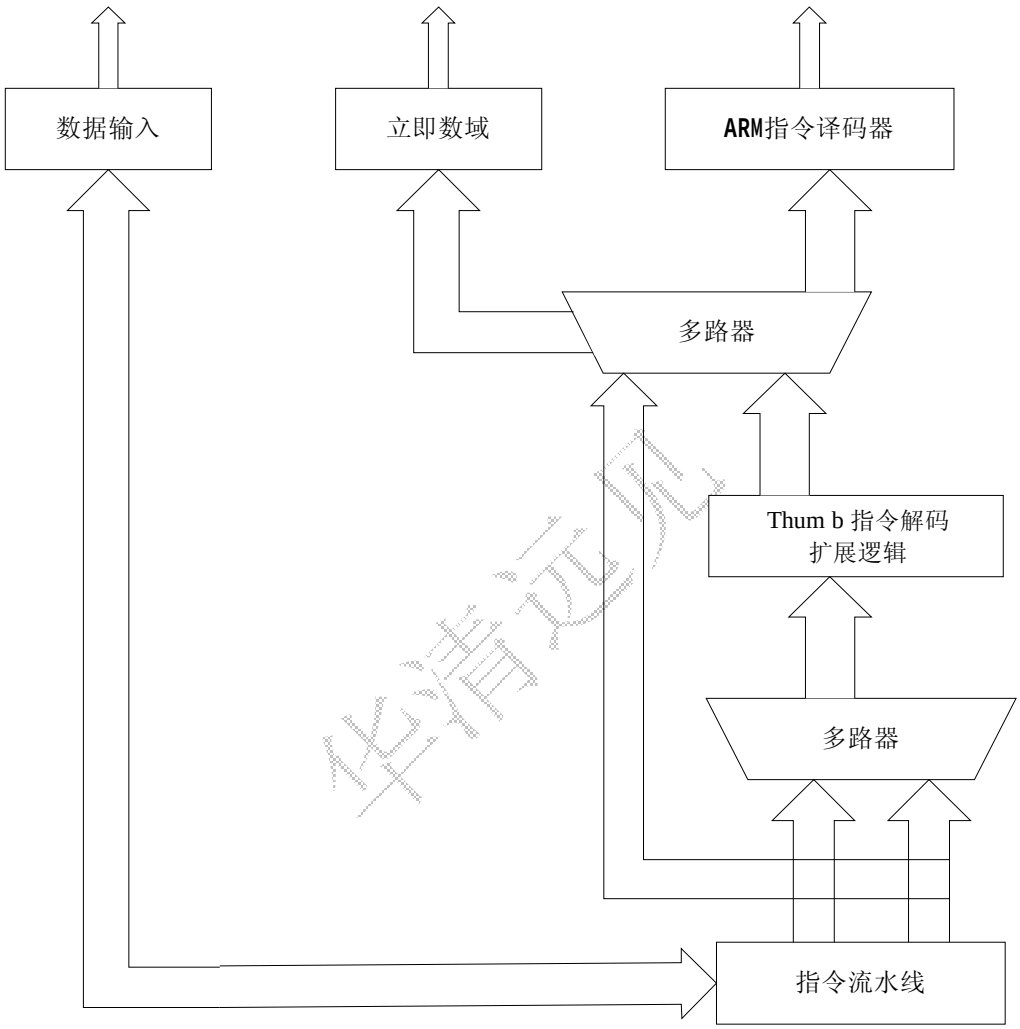


图 11.66 Thumb 指令的扩展逻辑组织

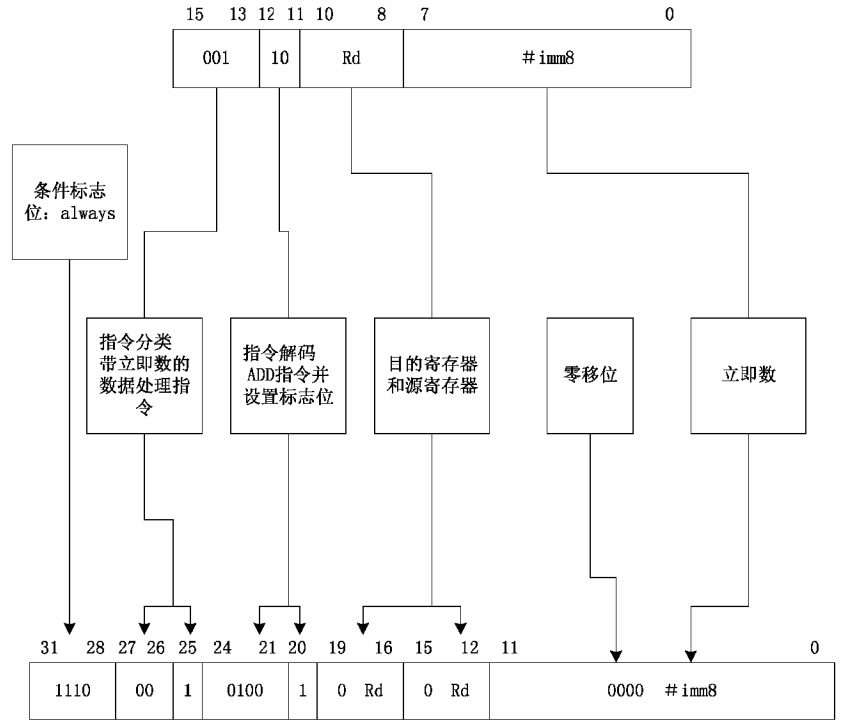


图 11.67　Thumb 指令到 ARM 指令的映射

Thumb 指令解码逻辑实现 Thumb 指令到 ARM 指令映射时遵循以下规则：

① ARM 指令的条件域（cond，bits[31：28]）默认使用 always。

注意　转移指令除外，转移指令是 Thumb 指令集中惟一一个条件执行的指令。

② 在 Thumb 指令操作码中隐含地指定 Thumb 数据处理指令是否应该修改 CPSR 中的条件标志位，在 ARM 指令中要明确指定。

③ 通过重复寄存器指示符将 Thumb 指令的 2 地址指令格式转换为 ARM 的 3 地址指令格式。

Thumb 指令解码扩展逻辑的简单性对 Thumb 指令集的效率是非常重要的。如果 Thumb 指令解码扩展逻辑构成复杂、速度低并且功耗大，那么 Thumb 指令就没有什么价值了。

11.9.2　Thumb 的特点

Thumb 指令把 32 位的 ARM 指令集的一个子集进行编码，成为一个 16 位的指令集。所以在 16 位的外部数据总线宽度下，在 ARM 处理器上使用 Thumb 指令的性能要比使用 ARM 指令性能更好；而在 32 位外部数据总线宽度下，使用 Thumb 指令的性能要比使用 ARM 指令的性能差。

16 位的 Thumb 指令只用 ARM 指令一半的位数来实现同样的功能，所以 Thumb 指令比 ARM 指令语义内涵少，因而实现特定程序所需要的 Thumb 指令数比 ARM 指令多。一般情况下，Thumb 代码所需要的空间为 ARM 代码的 70%。

注意 代码密度是 Thumb 指令集的主要优势。由于 Thumb 指令集的设计是面向编译器的，而不是针对手写汇编的，所以推荐使用高级语言如 C 或 C++语言来编程，然后用编译器生成 Thumb 指令的目标码。

下面的例子分别使用 ARM 指令和 Thumb 指令实现了同样的除法运算。虽然 Thumb 指令的实现使用了更多的指令，但是它所占的总的存储空间却比较少。

ARM 代码：

```
    MOV  r0，＃10
    MOV  r1，＃4
    MOV  r3，＃0
LOOP
    SUBS  r0，r0，r1
    ADDGE  r3，r3，＃1
    BGE  LOOP
    ADD  r2，r0，r1
```

Thumb 代码：

```
    MOV  r0，＃10
    MOV  r1，＃4
    MOV  r3，＃0
LOOP
    ADD  r3，＃1
    SUB  r0，r1
    BGE  LOOP
    SUB  r3，＃1
    ADD  r2，r0，r1
```

上面的 ARM 和 Thumb 代码分别实现了同样的除法运算。程序中分别使用 r0 和 r1 作为输入寄存器分别存放被除数和除数，而 r2 和 r3 用来存放指令的执行结果，分别为除法的余数和商。例子中 ARM 指令所占的存储空间为 7×4=28 字节，而 Thumb 指令所占存储空间为 8×2=16 字节。

从上例中不难看出，虽然实现特定程序所需要的 Thumb 指令数比 ARM 指令多，但 Thumb 指令所占用的内存空间要小于 ARM 指令。

当程序在选择使用 ARM 指令或 Thumb 指令时，要注意参考以下 Thumb 指令的特点。

① Thumb 代码所需空间为 ARM 代码的 70%。

② Thumb 代码使用的指令数比 ARM 代码多 40%。

③ 使用 16 位的存储器，Thumb 代码比 ARM 代码快 40%。

④ 使用 Thumb 代码，外部存储器的功耗比 ARM 代码少 30%。

因此，若性能更重要，则系统应使用 32 位存储器并运行 ARM 代码；如果成本及功耗更

重要，则最好选择 16 位的 Thumb 代码。如果能在同一个工程中，将两者结合使用，会在两方面取得最好的结果：在高端的 32 位 ARM 系统可以使用 Thumb 代码实现特定的非关键程序，以节省功耗或降低对存储器的要求；低端的 16 位系统可以使用小规模的 32 位片上 RAM 运行 ARM 代码的关键程序，所有非关键程序使用片外 Thumb 代码。

11.10 ARM 和 Thumb 的混合编程

11.10.1 互交工作基础

Thumb 以其较高的代码密度和在窄存储器上的性能，使得它在很多系统中得到广泛应用。但在很多情况下，还是不得不使用 ARM 指令，这是因为：

① ARM 代码比 Thumb 代码有更快的执行速度；

② ARM 处理器的一些特定功能必须由 ARM 指令实现，其中包括 PSR 指令、协处理器指令；

③ 异常发生时，处理器自动进入 ARM 状态，如果异常处理程序需要使用 Thumb 指令也必须通用一个 ARM 程序头（ARM assembler header）。

基于以上原因，即使程序需要由 Thumb 代码实现，也必须通过 ARM-Thumb 互交（ARM-Thumb interworking）进入 Thumb 状态。

ARM-Thumb 互交是指对汇编语言和 C/C++语言的 ARM 和 Thumb 代码进行连接的方法，它进行两种状态（ARM 和 Thumb 状态）间的切换。在进行这种切换时，有时需使用额外的代码，这些代码被称为 Veneer。AAPCS 定义了 ARM 和 Thumb 过程调用的标准。

从一个 ARM 例程调用一个 Thumb 例程，内核必须进行状态切换。状态的变化由 CPSR 的 T 位来显示。在跳转到一个例程时 BX 指令可用于 ARM 和 Thumb 状态切换，具体用法如下。

在 Thumb 状态调用 ARM 例程时，采用：

```
BX  Rn;
```

在 ARM 状态调用 Thumb 例程时，采用：

```
BX{cond}  Rn;
```

其中，Rn 可以是 r0～r15 中的任意寄存器。

这种带状态切换的跳转指令 BX，将寄存器 Rn 的内容拷贝到程序计数器寄存器 PC，因此可以实现 4G 空间的跳转。指令根据寄存器 Rn 的 bit[0]来决定处理器是否进行状态切换，详细内容参见 ARM 指令一节。

下面是一段 ARM 程序，该程序调用虚拟的 SWI_writeC 子程序从存储器的固定地址取出字符串“hello world”并输出。

```
        AREA    Hello,CODE,READONLY
SWI_WriteC       EQU       &0          ;软中断调用参数
```

```
SWI_Exit          EQU       &11        ;程序退出软中断调用参数
       ENTRY
START    ADR    r1,TEXT                ;取字符串地址
LOOP    LDRB    r0,[r1],#1             ;取下一字节内容
        CMP  r0,#0                     ;判断是否为字符串尾
        SWINE  SWI_WriteC              ;软中断调用打印字符
        BEN  LOOP                      ;循环
        SWI  SWI_Exit                  ;软中断调用退出程序执行
TEXT     =      “Hello World” ,&0a,&0d,0
        END
```

下面的代码将上面的 ARM 代码转换成等价的 Thumb 代码。

```
        AREA  HelloW_Thumb,CODE,READONLY
SWI_WriteC  EQU    &0              ;软中断调用参数
SWI_Exit  EQU    &11               ;程序退出软中断调用参数
        ENTRY                      ;程序入口点
        CODE32  进入 ARM 状态
        ADR r0, START+1            ;取得 Thumb 代码入口地址
        BX r0                      ;进入 Thumb 代码
        CODE16                     ;Thumb 代码入口点
START   ADR  r1, TEXT              ;r1 -> "Hello World"
LOOP    LDRB  r0, [r1]             ;取下一字节内容
        ADD  r1, r1, #1            ;地址指针加 1  **T
        CMP r0, #0                 ;判断是否为字符串尾
        BEQ DONE                   ;完成? **T
        SWI SWI_WriteC             ;如果不是字符串尾
        B LOOP                     ;继续循环
DONE    SWI  SWI_Exit              ;程序退出
        ALIGN                      ;字对齐
TEXT    DATA
             "Hello World",&0a,&0d,&00
        END
```

上例中，ARM 代码到 Thumb 代码转换过程中新增加的指令用“**T”标注。

在实现 ARM 代码和 Thumb 代码转换时，大部分的 ARM 指令有等价的 Thumb 指令，只有少数指令没有。如加载字节指令（LDR）不支持自动变址，软中断指令不能条件执行。

在编写 Thumb 代码时要注意以下几点。

① 汇编器需要知道什么时候产生 ARM 代码、什么时候产生 Thumb 代码，程序中使用 CODE32 和 CODE16 伪操作提供给编译器这些信息。

② 由于处理器上电执行是在 ARM 状态下完成的，所以要使用 Thumb 指令必须由 ARM 指令调用 Thumb 指令，这一过程是通过“BX　LR”指令来实现的。需要注意的是，在使用“BX　LR”指令前，要对寄存器 LR 做正确的初始化。

③ 在 ARM 和 Thumb 混合编程时，常使用 ALIGN 伪操作保证内存地址对齐。

11.10.2　互交子程序

编写 ARM/Thumb 互交代码时，下面两点需要注意。

① 对于 C/C++子程序而言，只要在编译时指定--apcs/interwork 选项，汇编器会生成合适的返回代码，使得程序返回到和调用程序相同的状态。

② 在汇编语言子程序中，用户必须自己编写相应的返回代码，使得程序返回到和调用程序相同的状态。

如果目标代码包含以下内容，应该在编译或汇编时使用--apcs/interwork 选项使处理器能够在 ARM 和 Thumb 代码间进行正确的切换，这种情况包含以下 4 种。

① 需要返回到 ARM 状态的 Thumb 子程序。

② 需要返回到 Thumb 状态的 ARM 子程序。

③ 间接调用 ARM 子程序的 Thumb 子程序。

④ 间接调用 Thumb 子程序的 ARM 子程序。

如果在程序连接阶段，连接器发现 ARM 子程序和 Thumb 子程序间存在相互调用，而源文件在编译时没有使用--apcs/interwork 选项，则连接器将报告以下错误。

```
Error: L6239E: Cannot call ARM symbol 'arm_function' in non-interworking object
armsub.o from THUMB code in thumbmain.o(.text)
```

其中，“arm_function”为需要进行状态切换的子程序名。

在这种情况下，用户必须使用--apcs/interwork 选项重新对源文件进行编译。

但在下面两种情况下，不必指定--apcs/interwork 选项。

① 在 Thumb 状态下，发生异常中断时，处理器自动切换到 ARM 状态，这时不需要添加状态切换代码。

② 当异常发生在 Thumb 状态时，从异常返回不需要添加状态切换的 Veneer 代码。

1．使用汇编语言实现互交

对于汇编程序来说，可以有两种方法来实现程序状态的切换。第一种方法是利用连接器提供的交互子程序 Veneer 来实现程序状态的切换，这时用户可以使用指令 BL 来调用子程序；另一种方法是用户自己编写状态切换的程序。

在 ARMv4 版本及其以前的版本中，可以使用 BX 指令实现程序状态的切换。

从 ARMv5 版本开始，下面的指令也可以用来实现程序的状态切换。

- BX（Branch and eXchange）
- BLX、LDR、LDM 和 POP

下面的两个伪操作用来区分源程序中的 ARM 代码和 Thumb 代码。

```
• CODE16
• CODE32
```

下面简单介绍用于状态切换的指令和伪操作，更详细的信息请分别参见相关章节。

（1）BX 指令

ARM 状态下的 BX 指令，使程序跳转到指令中指定的参数 Rm 指定的地址执行程序，Rm 的第 0 位拷贝到 CPSR 中 T 位，位[31：1]移入 PC。若 Rm 的 bit[0]为 1，则跳转时自动将 CPSR 中的标志位 T 置位，即把目标地址的代码解释为 Thumb 代码；若 Rm 的位 bit[0]为 0，则跳转时自动将 CPSR 中的标志位 T 复位，即把目标地址代码解释为 ARM 代码。指令的语法格式如下。

```
BX{<cond>}  <Rm>
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rm>

包含跳转指令的目标地址。如果 Rm 的 bit[0]=0，目标地址处指令为 ARM 指令；如果 Rm 的 bit[0]=1，目标地址处指令为 Thumb 指令。

指令操作的伪代码。

指令操作的伪代码如下面程序段所示。

```
If  conditionPassed{cond}  then
    T  Flag=Rm[0]
    PC = Rm AND 0xfffffffe
```

Thumb 状态下的 BX 指令，用于 ARM 和 Thumb 代码间的相互调用。

指令的语法格式。

```
BX  <Rm>
```

其中<Rm>为目标地址寄存器，包含程序的跳转地址。BX 指令的目标地址寄存器可以是 r0～r15 中的任意寄存器。

注意 如果 Rm[1: 0]=0b10，不满足 ARM 指令的内存对齐方式。指令的执行结果不可预知。如果该指令使用 r15 作为目标寄存器，其操作方式和使用其他寄存器相同。

指令操作的伪代码如下所示。

```
T  Flag=Rm[0]
PC=Rm[31: 1]<<1
```

ARM 指令集中的 BX 指令和 Thumb 指令集中的 BX 指令相差较大，它们分别为不同方向的跳转。当 r15 作为目的寄存器使用时，要特别注意该指令在两个指令集中的区别。

（2）BLX 指令

ARM 状态下的 BLX 指令使用一个寄存器中的绝对地址或标号，用于使程序跳转到 Thumb 状态或从 Thumb 状态返回，该指令用分支寄存器的最低位来更新 CPSR 中的 T 位，并将返回地址写入到连接寄存器 LR 中。

指令的语法格式如下所示：

```
① BLX  <target_addr>
② BLX{<cond>}  <Rm>
```

第一种格式中，转移目标按下述方法计算。将指令中指定的 24 位偏移量进行符号扩展，左移两位形成字偏移量，然后将其累加进程序计数器 PC 中。这时，程序计数器的内容为 BX 指令地址加 8 字节。位 H（bit[24]）也加到结果地址的第一位（bit[1]），使目标地址为半字地址，以执行接下来的 Thumb 指令。计算偏移量的工作一般由 ARM 汇编器来完成。这种形式的跳转指令只能实现±32MB 空间的跳转。

第二种格式中，寄存器 Rm 指定转移目标，Rm 的第 0 位拷贝到 CPSR 中的 T 位，bit[31：0]移入 PC。

- 如果 Rm 的 bit[0]=1，则跳转时自动将 CPSR 中的标志位 T 置位，即把目标地址的代码解释为 Thumb 代码。
- 如果 Rm 的 bit[0]=0，则跳转时自动将 CPSR 中的标志位 T 复位，即把目标地址的代码解释为 ARM 代码。

指令操作的伪代码如下面程序段所示。

第一种格式 BLX 指令。

```
LR=address of the instruction after the BLX instruction
T Flag=1
PC=PC + PC = PC + (SignExtend(signed_immed_24)<<2) + (H<<1)
```

第二种格式 BLX 指令。

```
If  ConditionPass{cond}  then
      LR = address of the instruction after the branch instruftion
      T Flag=Rm[0]
      PC=Rm AND 0xfffffffe
```

Thumb 状态下带返回链接的跳转指令 BLX（1）提供了一种在 Thumb 状态下无条件调用 ARM 子程序的方法，当从子程序返回时，通常使用下面的方式之一：

- BX LR
- 加载 PC 的 LDR 或 LDM 指令。

BLX 指令不可条件执行，可以实现在大约±4MB 的地址空间范围内跳转，实现方法是将一条 BLX 指令编译成两条 16 位的 Thumb 指令，从而实现上述跳转。对编译后的两条指令说明如下。

① H=10 的跳转指令。该跳转包含跳转偏移量的高位部分。

② H=01 的跳转指令。该跳转包含跳转偏移量的低位部分。

指令的语法格式

```
BL  <target_address>
```

<target_address>

指定程序跳转的目标地址。指令通过下面的方法计算目标地址。

- 将 H=10 的 BL 指令的 offset_11 域左移 12 位。
- 将结果符号扩展为 32 位。
- 将得到的值加到 PC 寄存器中。
- 与 H=11 的 BL 指令的 offset_11 域相加。

因此 BL 指令可以实现在大约±4MB 的地址空间范围内跳转。

指令操作的伪代码为：

```
if  H==10  then
    LR=PC+(SignExtend(offset_11)<<12)
Else  if  H==11  then
    PC=LR+(offset_11<<11)
    LR=(address  of  next  instruction)|1
Else  if  H==01  then
    PC=(LR+(offset_11<<1)) AND 0xFFFFFFFC
    LR=(address of next instruction)|1
Else  if  H==01  then
    PC=(LR+(offset_11<<1)) AND 0Xfffffffc
    LR=(address of next instruction)|1
    T Flag=0
```

另外 Thumb 状态下包含另一种格式的 BLX 指令，该 BLX（2）指令用于 ARM 和 Thumb 子程序间的相互调用。程序状态字的 T 标志位根据目的寄存器的 bit[0]位而改变。

指令的语法格式为：

```
BLX  <Rm>
```

其中<Rm>为目标地址寄存器，r0～r14 寄存器均可以作为目标地址寄存器。

注意 如果在此指令中使用 r15 作为目的寄存器，指令的执行结果不可预知。

此指令只在 ARMv5 版本以上指令集中被支持。

指令操作的伪代码为：

```
LR = (address of the instruction after this BLX)|1
T Flag = Rm[0]
PC = Rm[31:1]<<1
```

（3）汇编伪指令

汇编编译器可以产生 ARM 代码也可以产生 Thumb 代码。使用--thumb 或--16 选项指示编译器产生 Thumb 代码。由于所有支持 Thumb 代码的 ARM 处理器都从 ARM 状态开始执行，所以必须人为地使用 BX 或 BLX 指令，使处理器状态切换到 Thumb 状态并使用下面的伪操作使编译器产生 Thumb 代码。

- CODE16
- CODE32

CODE32 伪操作指示汇编器后面的指令为 32 位的 ARM 指令。

ARM 和 CODE32 伪操作的意义相同。

当汇编器对源程序进行编译时，如果需要，将会在程序中插入空指令，以保证内存单元字对齐。

语法格式如下。

```
ARM
CODE32
```

使用在同时包含 ARM 指令和 Thumb 指令的源文件中。当需要从 ARM 指令序列切换到 Thumb 指令序列时，使用伪操作 ARM（或 CODE32）；当需要从 Thumb 指令序列切换到 ARM 指令序列时使用 Thumb 伪操作。ARM（或 CODE32）伪操作只是指示汇编器后面的指令类型是 ARM 指令。该伪操作本身并不进行程序状态的切换，要进行状态切换，可以使用 BX 指令操作。

CODE16 伪指令通知编译器，其后的指令序列为 16 位的 Thumb 指令。

语法格式如下。

```
CODE16
```

若在汇编源程序中同时包含 ARM 指令和 Thumb 指令时，可用 CODE16 伪指令通知编译器其后的指令序列为 16 位的 Thumb 指令。

（4）编程实例

```
     PRESERVE8
     AREA    AddReg,CODE,READONLY ;段名为 AddReg，属性为 READONLY
     ENTRY                        ;程序入口
; SECTION 1
main
     ADR r0, ThumbProg + 1        ;确定跳转地址
                                  ;并将 bit[0]置 1
                                  ;使程序切换到 thumb 状态
     BX  r0                       ;程序跳转并执行状态切换
; SECTION 2
     CODE16                       ;Thumb 代码指示伪操作
ThumbProg
```

```
        MOV r2, #2                  ;r2 = 2
        MOV r3, #3                  ;r2 = 3
        ADD r2, r2, r3              ;r2 = r2 + r3
        ADR r0, ARMProg
        BX  r0                      ;程序跳转并执行状态切换
    ; SECTION 3
        CODE32                      ;ARM 代码指示伪操作
    ARMProg
        MOV r4, #4
        MOV r5, #5
        ADD r4, r4, r5

    ; SECTION 4
    stop MOV r0, #0x18              ;设置 semihosting 软中断号
        LDR r1, =0x20026            ;ADP_Stopped_ApplicationExit
        SWI 0x123456                ;ARM semihosting SWI 软中断调用
        END                         ;文件结束
```

上面的例子分为 4 部分，通过下面的步骤编译和运行。

① 使用文本编辑器，如 notepad，输入上面的代码，并保存成文件 addreg.s；

② 在命令行中键入汇编命令 armasm –g addreg.s；

③ 在命令行中键入链接命令 armlink addreg.o -o addreg；

④ 使用调试器（Debugger）（如，RealView Debuggeror AXD）运行映像文件。可以使用单步执行，观察代码在 Thumb 状态下的执行。

注意 Thumb 代码的地址标号如果用伪操作 export 声明为“外部的”，则连接器会自动调整该地址标号使其 bit[0]等于 1；如果该地址标号没有被声明为“外部的”，则使用者必须手动地对标号进行调整，如上例中的 ThumProg+1。

（5）ARMv5 架构下的状态切换

在 ARMv5 体系结构的指令集中，增加了下面两条指令用于 ARM 和 Thumb 代码互交。

- BLX address

该指令跳转到指令中指定的地址处执行程序并进行程序状态切换，该地址是“PC 相关的”，地址范围为±32MB（ARM 状态）或±4MB（Thumb 状态）。

- BLX register

在该中格式的跳转指令中寄存器 Rm 指定转移目标，Rm 的第 0 位拷贝到 CPSR 中的 T 位，bit[31：0]移入 PC。

使用上面两条指令，在执行程序跳转之前，处理器自动将返回连接寄存器 LR 的 bit[0]位更新为 CPSR 寄存器的 T 位，所以无论处理器状态是否发生变化，程序都能正确返回。

当使用 LDR、LDM 及 POP 指令向 PC 寄存器中赋值时，寄存器 CPSR 中的 Thumb 位将

被设置成 PC 寄存器的 bit[0]，这时就实现了程序状态的切换。这种方法在子程序的返回时非常有效，同样的指令可以根据需要返回到 ARM 状态或 Thumb 状态。

连接器在对目标代码进行连续时，将代码中的地址标号分为 3 类。

- ARM 指令地址标号。
- Thumb 指令地址标号。
- 数据（Data）地址标号。

当连接器重定位 Thumb 代码中的地址标号时，地址标号的 bit[0]位将被自动设置为 1。这就意味着跳转指令（这些指令包括 BX、BLX 和 LDR）可以根据目标地址正确的进行状态切换。

注意 上面提到的连接器自动设置目的地址的行为，只有在 ARMv5 及其以上版本中支持。

2．使用 C 和 C++语言实现互交

对于不同的 C 和 C++源程序，可以有些程序中包含 ARM 指令，有些程序中包含 Thumb 指令，这些程序可以相互调用，只是在编译这些程序时指定--apcs/interwork 选项。当使用了--apcs/interwork 选项，编译器会自动进行一些相应处理；连接器在检测到程序中存在互交工作时，会生成一些用于程序状态切换的代码。

（1）代码编译

可以使用下面的指令，将 C 或 C++程序编译为可以执行互交的目标代码。

```
armcc --c90 --thumb --apcs /interwork
armcc --c90 --arm --apcs /interwork
armcc --cpp --thumb --apcs /interwork
armcc --cpp --arm --apcs /interwork
```

注意 --cpp 是 C++文件（文件后缀为.cpp）默认的编译选项。

使用--apcs/interwork 选项对文件进行编译时，编译器会进行如下处理。

- 对于叶子程序（leaf function，即程序中没有其他子程序调用的程序），编译器将程序中的“MOV　PC，LR”指令替换成“BX　LR”指令，因为“MOV　PC”指令不能进行状态切换。
- 对于非叶子程序，要进行一系列的指令替换，如：

```
POP  {r4,r5,pc}
```

替换为：

```
  POP  {r4,r5}
  POP  {r3}
  BX   r3
```

下面的例子显示了一段带子程序调用的 C 语言程序，使用--apcs/interwork 选项进行编译时，对代码产生的影响。

C 语言源程序。

```
Void  func(void)
{
….
Sub()

..
}
```

使用 armcc --apcs/interwork 选项进行编译产生结果如下。

```
Func
STMFD  sp!,{r4-r7,lr}
….
BL  sub
….
LDMFD  sp!, {r4-r7,lr}
BX  lr
```

使用 tcc --apcs/interwork 选项进行编译产生结果如下。

```
PUSH  {r4-r7,lr}
….
BL  sub
….
POP  {r4-r7}
POP  {r3}
BX
```

（2）C 语言的互交实例

下面的例子显示了一个 Thumb 状态下的代码通过互交调用 ARM 子程序；而后又在 ARM 子程序中调用 Thumb 指令集的库函数 printf（）。

```
/*********************
*      thumbmain.c  *
*********************/
#include <stdio.h>
extern void arm_function(void);
int main(void)
{
```

```
        printf("Hello from Thumb World\n");
        arm_function();
        printf("And goodbye from Thumb World\n");
        return (0);
    }
    /*********************
    *          armsub.c    *
    **********************/
    #include <stdio.h>
    void arm_function(void)
    {
        printf("Hello and Goodbye from ARM world\n");
    }
```

使用下面的命令对程序进行编译连接。

① 编译生成带互交的 Thumb 代码。

```
armcc --thumb -c -g --apcs /interwork -o thumbmain.o thumbmain.c
```

② 编译生成带互交的 ARM 代码。

```
armcc -c -g --apcs /interwork -o armsub.o armsub.c
```

③ 连接目标文件。

```
armlink thumbmain.o armsub.o -o thumbtoarm.axf
```

另外，可以使用--info 选项使连接器输出由于互交所增加的代码大小。

```
armlink armsub.o thumbmain.o -o thumbtoarm.axf --info veneers
```

输出信息如下所示。

```
Adding Veneers to the image
    Adding TA veneer(4 bytes, Inline) for call to'arm_function'from
thumbmain.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__0printf' from armsub.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__rt_lib_init' from
kernel.o(.text).
    Adding AT veneer (12 bytes,Long) for call to'__rt_lib_shutdown'from
kernel.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_memclr_w' from
stdio.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_raise' from stdio.o(.text).
```

```
        Adding TA veneer (8 bytes, Short) for call to '__rt_exit' from exit.o(.text).
        Adding TA veneer (4 bytes, Inline) for call to '__user_libspace' from
free.o(.text).
        Adding TA veneer (4 bytes, Inline) for call to '_fp_init' from
lib_init.o(.text).
        Adding TA veneer (4 bytes, Inline) for call to '__heap_extend' from
malloc.o(.text).
        Adding AT veneer (8 bytes, Inline) for call to '__raise' from rt_raise.o(.text).
        Adding TA veneer (4 bytes, Inline) for call to '__rt_errno_addr' from
ftell.o(.text).
    12 Veneer(s) (total 72 bytes) added to the image.
```

（3）Thumb 状态下的功能指针

任何指向 Thumb 函数（由 Thumb 指令完成的功能函数并且其返回状态也为 Thumb 状态）的指针，其最低有效位（LSB）必为 1。

当重定位 Thumb 代码中的地址标号时，连接器将自动设置地址的最低有效位。如果在程序中使用绝对地址，连接器将无法完成该设置。因此，如果在 Thumb 代码中使用绝对地址时，必须手工设置为其地址加 1。

下面的例子显示了 Thumb 代码的功能指针的使用。

```
    typedef int (*FN)();
    myfunc() {
        FN fnptrs[] = {
            (FN)(0x8084 + 1),          // 有效的 Thumb 地址
            (FN)(0x8074)               // 无效的 Thumb 地址
        };
        FN* myfunctions = fnptrs;
        myfunctions[0]();              // 调用成功
        myfunctions[1]();              // 调用失败
    }
```

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 12 章 混合使用 C、C++和汇编语言

本章主要描述如何实现 C、C++和汇编语言的混合编程。重点讲解了如何在 C 和 C++中使用内联（inline）汇编和嵌入型（embedded）汇编。本章内容主要包括：

- 内联汇编和嵌入型汇编的使用；
- 如何在汇编语言中使用 C 变量；
- 如果在 C++文件中包含 C 头文件；
- 如果实现汇编程序、C 程序以及 C++程序间的相互调用。

12.1 内联汇编和嵌入型汇编的使用

内联汇编和嵌入型汇编是包含在 C/C++编译器中的汇编器。使用它可以在 C/C++程序中实现 C/C++语言不能完成的一些工作。例如，在下面几种情况中必须使用内联汇编或嵌入型汇编。

- 程序中使用饱和算术运算（Saturating arithmetic），如 SSAT16 和 USAT16 指令。
- 程序中需要对协处理器进行操作。
- 在 C 或 C++程序中完成对程序状态寄存器的操作。

使用内联汇编编写的程序代码效率也比较高。

12.1.1 内联汇编

1. 内联汇编语法

内联汇编使用“_asm”（C++）和“asm”（C 和 C++）关键字声明，语法格式如下所示。

```
• __asm("instruction[;instruction]");  // 必须为单条指令
  __asm{instruction[;instruction]}
• __asm{
  ...
  instruction
  ...
}
• asm("instruction[;instruction]");    // 必须为单条指令
  asm{instruction[;instruction]}
• asm{
   ...
   instruction
   ...
}
```

内联汇编支持大部分的 ARM 指令，但不支持带状态转移的跳转指令，如 BX 和 BLX 指令，详见 ARM 相关文档。

由于内联汇编嵌入在 C 或 C++程序中，所有在用法上有其自身的一些特点。

① 如果同一行中包含多条指令，则用分号隔开。

② 如果一条指令不能在一行中完成，使用反斜杠“/”将其连接。

③ 内联汇编中的注释语句可以使用 C 或 C++风格的。

④ 汇编语言中使用逗号“,”作为指令操作数的分隔符，所以如果在 C 语言中使用逗号必须用圆括号括起来。如，__asm {ADD x, y, (f(), z)}。

⑤ 内联汇编语言中的寄存器名被编译器视为 C 或 C++语言中的变量，所以内联汇编中

出现的寄存器名不一定和同名的物理寄存器相对应。这些寄存器名在使用前必须声明，否则编译器将提示警告信息。

⑥ 内联汇编中的寄存器（除程序状态寄存器 CPSR 和 SPSR 外）在读取前必须先赋值，否则编译器将产生错误信息。下面的例子显示了内联汇编和真正汇编的区别。

错误的内联汇编函数如下所示。

```
int f(int x)
{
      __asm
      {
          STMFD sp!, {r0}  // 保存 r0 不合法，因为在读之前没有对寄存器写操作
          ADD r0, x, 1
          EOR x, r0, x
          LDMFD sp!, {r0}  // 不需要恢复寄存器
      }
      return x;
}
```

将其进行改写，使它符合内联汇编的语法规则。

```
int f(int x)
{
      int r0;
      __asm
      {
          ADD r0, x, 1
          EOR x, r0, x
      }
      return x;
}
```

下面通过几个例子进一步了解内联汇编的语法。

① 字符串拷贝

下面的例子使用一个循环完成了字符串的拷贝工作。

```
#include <stdio.h>
void my_strcpy(const char *src, char *dst)
{
     int ch;
     __asm
     {
     loop:
```

```
            LDRB    ch, [src], #1
            STRB    ch, [dst], #1
            CMP     ch, #0
            BNE     loop
        }
    }
    int main(void)
    {
        const char *a = "Hello world!";
        char b[20];
        my_strcpy (a, b);
        printf("Original string: '%s'\n", a);
        printf("Copied string: '%s'\n", b);
        return 0;
    }
```

② 中断使能

下面的例子通过读取程序状态寄存器 CPSR 并设置它的中断使能位 bit[7]来禁止/打开中断。需要注意的是，该例只能运行在系统模式下，因为用户模式是无权修改程序状态寄存器的。

```
    __inline void enable_IRQ(void)
    {
        int tmp;
        __asm
        {
            MRS tmp, CPSR
            BIC tmp, tmp, #0x80
            MSR CPSR_c, tmp
        }
    }
    __inline void disable_IRQ(void)
    {
        int tmp;
        __asm
        {
            MRS tmp, CPSR
            ORR tmp, tmp, #0x80
            MSR CPSR_c, tmp
        }
```

```
}
int main(void)
{
     disable_IRQ();
     enable_IRQ();
}
```

③ 分隔符的计算

下面的例子计算两个整数数组中分隔符“,”的个数。该例子显示了如何在内联汇编中访问 C 或 C++语言中的数据类型。该例中的内联汇编函数 mlal()被编译器优化为一条 SMLAL 指令，可以使用-S –interleave 编译选项使编译器输出汇编结果。

```
#include <stdio.h>
/* change word order if big-endian */
#define lo64(a) (((unsigned*) &a)[0])  /* long long 型的低 32 位 */
#define hi64(a) (((int*) &a)[1])       /* long long 型的高 32 位 */
__inline __int64 mlal(__int64 sum, int a, int b)
{
#if !defined(__thumb) && defined(__TARGET_FEATURE_MULTIPLY)
     __asm
     {
       SMLAL lo64(sum), hi64(sum), a, b
     }
#else
     sum += (__int64) a * (__int64) b;
#endif
     return sum;
}
__int64 dotprod(int *a, int *b, unsigned n)
{
     __int64 sum = 0;
     do
        sum = mlal(sum, *a++, *b++);
     while (--n != 0);
     return sum;
}
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int b[10] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
int main(void)
{
```

```
        printf("Dotproduct %lld (should be %d)\n", dotprod(a, b, 10), 220);
        return 0;
    }
```

2．内联汇编中的限制

可以在内联汇编代码中执行的操作有许多限制。这些限制提供安全的方法，并确保在汇编代码中不违反 C 和 C++ 代码编译中的假设。

① 不能直接向程序计数器 PC 赋值。

② 内联汇编不支持标号变量。

③ 不能在程序中使用“.”或{PC}得到当前指令地址值。

④ 在 16 进制常量前加“0x”代替“&”。

⑤ 建议不要对堆栈进行操作。

⑥ 编译器可能会使用 r12 和 r13 寄存器存放编译的中间结果，在计算表达式值时可能会将寄存器 r0～r3、r12 及 r14 用于子程序调用。另外在内联汇编中设置程序状态寄存器 CPSR 中的标志位 NZCV 时，要特别小心，内联汇编中的设置很可能会和编译器计算的表达式的结果冲突。

⑦ 用内联汇编代码更改处理器模式是可能的。然而，更改处理器模式会禁止使用 C 或 C++操作数或禁止对已编译 C 或 C++代码的调用，直到将处理器模式更改回原设置之后之前的函数库才可正常使用。

⑧ 为 Thumb 状态编译 C 或 C++时，内联汇编程序不可用且不汇编 Thumb 指令。

⑨ 尽管可以使用通用协处理器指令指定 VFP 或 FPA 指令，但内联汇编程序不为它们提供直接支持。

不能用内联汇编代码更改 VFP 向量模式。内联汇编可包含浮点表达式操作数，该操作数可使用编译程序生成的 VFP 代码求出操作数值。因此，仅由编译程序修改 VFP 状态很重要。

⑩ 内嵌汇编不支持的指令有 BX、BLX、BXJ 和 BKPT 指令。而 LDM、STM、LDRD 和 STRD 指令可能被等效为 ARM LDR 或 STR 指令。

3．内联汇编中的虚拟寄存器

内联汇编程序提供对 ARM 处理器物理寄存器的非直接访问。如果在内联汇编程序指令中将某个 ARM 寄存器用作操作数，它就成为相同名称的虚拟寄存器的引用，而不是对实际物理 ARM 寄存器的引用。例如内联汇编指令中使用了寄存器 r0，但对于 C 编译器，指令中出现的 r0 只是一个变量，并非实际的物理寄存器 r0，当程序运行时，可能是由物理寄存器 r1 来存放 r0 所代表的值。

下面的例子显示了编译器如何对内联汇编指令的寄存器进行分配。

程序的源代码如下。

```
#include <stdio.h>
void test_inline_register(void)
```

```
{
     int i;
     int r5,r6,r7;
     __asm
     {
         MOV  i,#0
     loop:
         MOV  r5,#0
         MOV  r6,#0
         MOV  r7,#0
         ADD  i,i,#1
         CMP  i,#3
         BNE  loop

     }
}

int main(void)
{

     test_inline_register ();
     printf("test inline register\n");

     return 0;
}
```

由 C 编译器编译出的汇编码如下所示。

```
    test_inline_register:
    0000807C E3A00000  MOV     r0,#0
>>> TEST_INLINE_REGISTER\#12    loop:
    00008080 E1A00000  NOP
>>> TEST_INLINE_REGISTER\#13        MOV  r5,#0
    00008084 E3A01000  MOV     r1,#0
>>> TEST_INLINE_REGISTER\#14        MOV  r6,#0
    00008088 E3A02000  MOV     r2,#0
>>> TEST_INLINE_REGISTER\#15        MOV  r7,#0
    0000808C E3A03000  MOV     r3,#0
>>> TEST_INLINE_REGISTER\#16        ADD  i,i,#1
    00008090 E2800001  ADD     r0,r0,#1
```

```
>>> TEST_INLINE_REGISTER\#17        CMP  i,#3
    00008094 E3500003  CMP     r0,#3
    00008098 0A000000  BEQ     0x80a0            <TEST_INLINE_REGISTER\#21>
>>> TEST_INLINE_REGISTER\#18        BNE  loop
    0000809C EAFFFFF8  B       0x8084          <TEST_INLINE_REGISTER\#13>
>>> TEST_INLINE_REGISTER\#21 }
    000080A0 E12FFF1E  BX      r14
>>> TEST_INLINE_REGISTER\#25 {
```

注意 下面的代码是由 Realview2.2 编译出的代码，使用其他编译器结果可能有差异。同一段内嵌汇编经过不同版本的编译器编译后，在指令里可能使用不一样的实际寄存器，但是只要遵循文档里的编码指导，执行的功能肯定相同。

例子中以“>>>”的开头的行是程序的源码部分，紧接其后的是由编译器编译出的汇编代码。从上例可以很清楚地看出，源程序中使用了 r5、r6 和 r7，但由编译器编译后的代码使用了寄存器 r1、r2 和 r3。

另外，需要特别指出的是在内联汇编中使用寄存器必须先声明其变量类型，如上例中的“int r5，r6，r7”。如果不在使用前进行声明，编译器将给出以下错误信息。

```
#1267-D: Implicit physical register R3 should be defined as a variable
```

编译程序定义的虚拟寄存器有函数局部作用范围，即在同一个 C 函数中，涉及相同虚拟寄存器名称的多个 asm 语句或声明，访问相同的虚拟寄存器。

内联汇编没有为 pc（r15）、lr（r14）和 sp（r13）寄存器创建虚拟寄存器，而且不能在内联汇编代码中读取或直接修改它们的值。如果内联汇编程序中出现了对这些寄存器的访问，编译器将给出以下错误消息。例如，如果指定 r14：

```
#20: identifier "r14" is undefined
```

内联汇编可以直接使用 CPSR 和 SPSR 对程序状态字进行操作，因为内联汇编中不存在虚拟处理器状态寄存器（PSR）。任何对 PSR 的引用总是指向物理 PSR。

4．内联汇编中的指令展开

内联汇编代码中的 ARM 指令可能会在编译过程中扩展为几条指令。扩展取决于指令、指令中指定的操作数个数以及每个操作数的类型和值。通常，被扩展的指令有以下两种情况：

- 含有常数操作的指令；
- LDM、STM、LDRD 和 STRD 指令；
- 乘法指令 MUL 被扩展为一系列的加法和移位指令。

下面的例子说明了编译器如何对含有常数操作的指令进行扩展。

包含有常数操作的加法指令：

```
ADD r0,r0,#1023
```

被编译器编译为如下两条指令：

```
ADD r0,r0,#1024
SUB r0,r0,#1
```

注意 扩展指令对程序状态寄存器 CPSR 的影响：算术指令影响相应的 NZCV 标准位；其他指令设置 NZ 标志位不影响 V 标志位。

所有的 LDM 和 STM 指令被扩展为等效的 LDR 和 STR 指令序列。然而，在优化过程中，编译程序可能因此将单独的指令重组为一条 LDM 或 STM 指令。

5．内联汇编中的常数

指令中的标志符“＃”是可选的（前面的例子中，指令中常数前均加了标志符“＃”）。如果在指令中使用了“＃”，则其后的表达式必为常数。

6．内联汇编指令对标志位的影响

内联汇编指令可能显式或隐式地更新处理器程序状态寄存器的条件标志位。在仅包含虚拟寄存器操作数或简单表达式操作数的内联汇编中，其执行结果是可以预见。如果指令中指定了隐式或显式更新条件标志位，则条件标志位根据指令的执行进行设置。如果未指定更新，则条件标志不会更改。如果内嵌汇编指令的操作数都不是简单操作数时或指令不显式更新条件标志位，则条件标志位可能会被破坏。一般情况下，编译程序不易诊断出对条件标志的潜在破坏。然而，在构造析构 C++临时函数的操作数时，如果指令试图更新条件标志，编译程序将给予警告，因为析构函数可能会破坏条件标志位。

7．内联汇编指令中的操作数

内联汇编指令中的操作数分为以下 4 种。

- 虚拟寄存器
- 表达式操作数
- 寄存器列表
- 中间操作数

（1）虚拟寄存器

在内联汇编指令中指定的寄存器表示虚拟寄存器而不是实际的物理寄存器。由编译器编译的汇编代码中使用的物理寄存器可能与在指令中指定的不同。每个虚拟寄存器的初值是不可预测的，必须在读取之前将初值写入虚拟寄存器。如果在写入之前试图读虚拟寄存器，编译程序会给予警告。

（2）表达式操作数

在内联汇编指令中，可将函数自变量、C 或 C++变量和其他 C 或 C++表达式指定为寄存器操作数。用作操作数的表达式必须为整数类型，如 char、short、int 或 long，（长整型 long long

除外）或指针类型。当表达式作为内联汇编指令的操作数时，编译器在编译时自动增加一段代码计算表示式的值并将其加载到指定的寄存器中。

注意 数据类型中除 char 和 short（默认为无符号类型）外，其他均为有符号类型。

下面的例子显示了编译器如何处理内联汇编中的表达式操作数。
程序源代码如下所示。

```
/* Example Operands */

void my_operand(void)
{
     int i,j,total;

     __asm
     {
     mov i,#0
     mov j,#1
     add total,j,i+j
     }
}

int main(void)
{

     my_operand ();

 }
```

由编译器编译出的汇编代码如下所示（其中只列出了内联汇编的一段代码）。

```
   my_operand:
   0000807C E3A01000  MOV     r1,#0
>>> OPERANDS\#12    mov j,#1
   00008080 E3A00001  MOV     r0,#1
   00008084 E0812000  ADD     r2,r1,r0
>>> OPERANDS\#13    add total,j,i+j
   00008088 E0803002  ADD     r3,r0,r2
>>> OPERANDS\#15 }
   0000808C E12FFF1E  BX      r14
>>> OPERANDS\#19 {
```

从编译的代码可以看出，编译器将“add total,j,i+j”分为两步来完成，用户在编写自己的内联汇编应用程序时要特别注意这一点。

包含多个表达式操作数的指令，没有指定表达式操作数求值的顺序。

将 C 或 C++表达式用作内联汇编程序操作数，如果表达式的值不能满足 ARM 指令中所要求的指令操作数约束条件，一条指令将被扩展为多条指令。

如果用作操作数的表达式创建需要析构的临时函数，析构将发生在执行内联汇编指令之后，这与 C++析构临时函数的规则相类似。

简单表达式操作数包含以下几种类型。

- 变量值
- 变量地址
- 指针变量的反引用（the dereferencing of a point varable）
- 伪操作指定的程序常量

非简单表达式操作数包含以下几种类型。

- 隐式函数调用，如除法，或显式函数调用
- C++临时函数的构造
- 算术或逻辑操作

（3）寄存器列表

寄存器列表最多可包含 16 个操作数。这些操作数可以是虚拟寄存器或表达式操作数。在寄存器列表中指定虚拟寄存器和表达式操作数的顺序非常重要。寄存器列表中操作数的读写顺序是从左到右。第一个操作数使用最低地址，随后的操作数的地址依次在前一地址基础上增加 4。这一点与 LDM 或 STM 指令的普通操作（编号最低的物理寄存器总是存入最低的存储器地址）是不同的。之所以存在这种区别是因为在内联汇编中使用的寄存器被编译器虚拟化了。

同一个表达式操作数或虚拟寄存器可以在寄存器列表中出现多次，重复使用。

如果表达式操作数或虚拟寄存器被指定为指令中的基址寄存器，表达式或虚拟寄存器的值按照 ARM 指令寻址方式进行更新。更新将覆盖原表达式或虚拟寄存器的值。

（4）中间操作数（Intermediate operands）

在内联汇编指令中，可能将 C 或 C++整型常量表达式用作立即数处理。用于指定直接移位的常量表达式的值必须在 ARM 指令规定的移位操作数的范围内；用于为存储器或协处理器数据传送指令指定直接偏移量的常量表达式，必须符合 ARM 体系结构中的内存对齐标准。

8．函数调用和分支跳转

利用内联汇编程序的 BL 和 SWI 指令可在常规指令字段后指定 3 个可选列表。这些指令格式有以下几种。

```
    SWI{cond}   swi_num   ,   {   input_param_list   },   {   output_value_list   },
{ corrupt_reg_list }
    BL{cond}     function,   {   input_param_list   },   {   output_value_list   },
{ corrupt_reg_list }
```

其中，swi_num 为 SWI 调用的中断号；function 为被调用函数名；{input_param_list}为输入参数列表；{output_value_list}为输出参数列表；{corrupt_reg_list}为被破坏寄存器列表。

注意 内联汇编程序不支持 BX、BLX 和 BXJ 指令。不能在任何输入、输出或“被破坏的寄存器列表（corrupted register list）”中指定 lr、sp 或 pc 寄存器；任何 SWI 指令或函数调用不能更改 sp 寄存器。

下面分别详细介绍语法格式中各参数的使用。

（1）未指定任何列表

如果在 SWI 和 BL 指令后没指定任何列表，则有下面规则。

- r0～r3 用作输入参数；
- r0 用于输出值；
- r12 和 r14 的值将会被修改。

（2）输入参数列表

指令中的输入参数列表{ input_param_list }列出了传递给被调用函数 function 和 SWI 的参数。被传递的参数可以是表达式、变量或包含表达式或变量的物理寄存器。

内联汇编编译器在编译时增加一小段编译程序负责在函数和 SWI 调用前将传递的参数载入特定的物理寄存器中。为确保与现有内联汇编代码的向后兼容性，程序中指定物理寄存器名称而并不对其赋值，使相同名称虚拟寄存器中的值出现在物理寄存器中。

例如，指令 BL foo {r0=expression1, r1=expression2, r2}生成以下伪代码：

```
MOV (physical) r0, expression1
MOV (physical) r1, expression2
MOV (physical) r2, (virtual) r2
BL foo
```

（3）输出参数列表

输出参数列表{ output_value_list }列出了用来存放功能函数和 SWI 调用返回值的寄存器或表达式。列表中的值可以是物理寄存器、可修改长值表达式或单个物理寄存器名称。

内联汇编程序从特定的物理寄存器中取值并赋值到特定的表达式中。指定物理寄存器名称而并不赋值，导致相同名称虚拟寄存器被物理寄存器中的值更新。

例如，BL foo { }, {result1=r0, r1}生成以下伪码：

```
BL foo
MOV result1, (physical) r0
MOV (virtual) r1, (physical) r1
```

（4）被破坏的寄存器列表（Corrupted register list）

此列表指定被函数调用破坏的物理寄存器。如果条件标志被调用的函数修改，必须在被破坏的寄存器列表中指定 PSR。

BL 和 SWI 指令总是破坏 lr。

如果指令中缺少此列表项，则 r0～r3、ip、lr 和 PSR 被破坏。

注意 指令 BL 和 B 的区别在于，跳转指令 B 只能使程序跳转到 C 或 C++程序的一个地址标号，不能用于子程序调用。

9．内嵌汇编中的标号

内联汇编代码中定义的标号可被用作跳转或 C 和 C++“goto”语句的目标。在内联汇编代码中，C 和 C++中定义的标号可被用作跳转指令的目标。

10．内嵌汇编器版本间的差异

不同版本的 ARM 编译器对内联汇编程序的语法要求有显著差异。在具体使用时请参见相关文档。

- 如果使用的是 ADS v1.2，请参阅 ADS 开发者指南；
- 如果使用的是 RVCT v1.2，请参阅 RealView 编译工具 1.2 版开发者指南。

12.1.2 嵌入式汇编

利用 ARM 编译器可将汇编代码包括到一个或多个 C 或 C++函数定义中去。嵌入式汇编器提供对目标处理器不受限制的低级别访问，利用它可以使用 C 和 C++预处理程序伪操作（preprocessor directive）并可以方便的使用偏移量访问结构成员。

本小节将介绍以下内容：

- 嵌入式汇编程序语法；
- 嵌入式汇编语句的限制；
- 嵌入式汇编程序表达式和 C 或 C++表达式之间的差异；
- 嵌入式汇编函数的生成；
- __cpp 关键字；
- 手动重复解决方案；
- 相关基类的关键字；
- 成员函数类的关键字；
- 调用非静态成员函数。

有关为 ARM 处理器编写汇编语言的详细信息，请参阅 ADS 或 RealView 编译工具的汇编程序指南。

1．嵌入式汇编语言语法

嵌入式汇编函数定义由 --asm（C 和 C++）或 asm（C++） 函数限定符标记，可用于：

- 成员函数；
- 非成员函数；
- 模板函数；
- 模板类成员函数。

用__asm 或 asm 声明的函数可以有调用参数和返回类型。它们从 C 和 C++中调用的方式与普通 C 和 C++函数调用方式相同。嵌入式汇编函数语法是：

```
__asm return-type function-name(parameter-list)
{
    // ARM/Thumb/Thumb－2 assembler code
    instruction[; instruction]
    ...
    [instruction]
}
```

嵌入式汇编的初始执行状态是在编译程序时由编译选项决定的。这些编译选项如下所示：

- 如果初始状态为 ARM 状态，则内嵌汇编器使用--arm 选项；
- 如果初始状态为 Thumb 状态，则内嵌汇编器使用--thumb 选项。

注意 嵌入式汇编的初始状态由编译器的编译选项确定，与程序中的#pragma arm 和 #pragma thumb 伪操作无关。

可以显示地使用 ARM、THUMB 和 CODE16 伪操作改变嵌入式汇编的执行状态。关于 ARM 伪操作的详细信息请参加指令伪操作一节。如果使用的处理器支持 Thumb-2 指令，则可以在 Thumb 状态下，在嵌入式汇编中使用 Thumb-2 指令。

参数名允许用在参数列表中，但不能用在嵌入式汇编函数体内。例如，以下函数在函数体内使用整数 i，但在汇编中无效：

```
__asm int f(int i) {
    ADD i, i, #1 // 编译器报错
}
```

可以使用 r0 代替 i。

下面通过嵌入式汇编的例子，来进一步熟悉嵌入式汇编的使用。

下面的例子实现了字符串的拷贝，注意和上一节中内联汇编中字符串拷贝的例子相比较，分析其中的区别。

```
#include <stdio.h>
__asm void my_strcpy(const char *src, const char *dst) {
loop
      LDRB  r3, [r0], #1
      STRB  r3, [r1], #1
      CMP   r3, #0
      BNE   loop
      MOV   pc, lr
}
void main()
{
```

```
    const char *a = "Hello world!";
    char b[20];
    my_strcpy (a, b);
    printf("Original string: '%s'\n", a);
    printf("Copied   string: '%s'\n", b);
}
```

2．嵌入式汇编语言的使用限制

嵌入式汇编的使用有下面一些限制。

① 在预处理之后，__asm 函数只能包含汇编代码，但以下标识符除外：

```
• __cpp(expr);
• __offsetof_base(D, B);
• __mcall_is_virtual(D, f);
• __mcall_is_in_vbase(D, f);
• __mcall_this_offset(D, f);
• __vcall_offsetof_vfunc(D, f);
```

② 编译程序不为__asm 函数生成返回指令。如果要从 __asm 函数返回，必须将用汇编代码编写的返回指令包含到函数体内。由于嵌入式汇编执行__asm 函数的顺序是在编译时定义好的，所有从一个内嵌汇编跳转到一个内嵌汇编程序是运行的，但在内联汇编中却不能实现。

③ __asm 函数调用遵循 AAPCS 规则。所以，即使在__asm 函数体内可用的汇编代码（例如，更改状态），在__asm 函数和普通 C 或 C++函数相互调用时，未必可用，因为此调用也必须遵循 AAPCS 规则。

3．嵌入式汇编程序表达式和 C 或 C++表达式之间的差异

嵌入式汇编表达式和 C 或 C++表达式之间存在以下差异。

① 汇编程序表达式总是无符号的。相同的表达式在汇编程序和 C 或 C++ 中有不同值。例如：

```
MOV r0, #(-33554432 / 2)        // 结果为 0x7f000000
MOV r0, #__cpp(-33554432 / 2)   // 结果为 0xff000000
```

② 以 0 开头的汇编程序编码仍是十进制的。例如：

```
MOV r0, #0700             // 十进制 700
MOV r0, #__cpp(0700)      // 八进制 0700 等于 十进制 448
```

③ 汇编程序运算符优先顺序与 C 和 C++ 不同。例如：

```
MOV r0, #(0x23 :AND: 0xf + 1)    // ((0x23 & 0xf) + 1) => 4
MOV r0, #__cpp(0x23 & 0xf + 1)   // (0x23 & (0xf + 1)) => 0
```

④ 汇编程序字符串不是以空字符为终止标志的：

```
DCB "no trailing null"                  // 16 bytes
DCB __cpp("I have a trailing null!!")   // 25 bytes
```

注意 在_cpp 标识符作用范围之内使用 C 或 C++语法规则。

4．嵌入式汇编函数的生成

由关键字__asm 声明的嵌入式汇编程序，在编译时将作为整个文件体传递给 ARM 汇编器。传递过程中，__asm 函数的顺序保持不变（用模板实例生成的函数除外）。正是由于嵌入式汇编的这个特性，使得由一个__asm 标识的嵌入式汇编程序调用在同一文件中的另一个嵌入式汇编程序是可以实现的。

当使用编译器 armcc 时，局部链接器（Partial Link）将汇编程序产生的目标文件与编译 C 程序的目标文件相结合，产生单个目标文件。

编译程序为每个 __asm 函数生成 AREA 命令。例如，以下__asm 函数：

```
#include <cstddef>
struct X { int x,y; void addto_y(int); };
__asm void X::addto_y(int) {
    LDR     r2,[r0, #__cpp(offsetof(X, y))]
    ADD     r1,r2,r1
    STR     r1,[r0, #__cpp(offsetof(X, y))]
    BX      lr
}
```

对于此函数，编译程序生成：

```
AREA ||.emb_text||, CODE, READONLY
EXPORT |_ZN1X7addto_yEi|
#line num "file"
|_ZN1X7addto_yEi| PROC
 LDR r2,[r0, #4]
 ADD r1,r2,r1
 STR r1,[r0, #4]
 BX lr
ENDP
END
```

由上面的例子可以看出，对于变量 offsetof 的使用必须加__cpp（）标识符才能引用，因为该变量是在 cstddef 头文件中定义的。

由__asm 声明的常规函数被放在名为.emb_text 的段（Section）中。这一点也是嵌入式汇

编和内联汇编最大的不同。相反，隐式实例模板函数（Implicitly Instantiated Template Function）和内联汇编函数放在与函数名同名的区域（Area）内，并为该区域增加公共属性。这就确保了这类函数的特殊语义得以保持。

由于内联和模板函数的区域的特殊命名，所以这些函数不按照文件中定义的顺序排列，而是任意排序。因此，不能以__asm 函数在原文件中的排列顺序，来判断它们的执行顺序，也就是说，即使两个连续排列的__asm 函数，也不一定能顺序执行。

5．关键字__cpp

可用__cpp 关键字从汇编代码中访问 C 或 C++的编译时常量表达式，其中包括含有外部链接的数据或函数地址。标识符__cpp 内的表达式必须是适合用作 C++静态初始化的常量表达式（请参阅 ISO/IEC 14882:1998 中的 3.6.2 非本地对象初始化一节和本书的常量表达式一节）。

编译时，编译器将使用__cpp(expr) 的地方用汇编程序可以使用的常量所取代。例如：

```
    LDR r0, =__cpp(&some_variable)
    LDR r1, =__cpp(some_function)
    BL  __cpp(some_function)
    MOV r0, #__cpp(some_constant_expr)
```

__cpp 表达式中的名称可在__asm 函数的 C++上下文中查阅。__cpp 表达式结果中的任何名称按照要求被损毁并自动为其生成 IMPORT 语句。

6．手动重复解决方案

可以在嵌入式汇编中使用 C++转换为非虚拟函数调用解决重复。例如：

```
    void g(int);
    void g(long);
    struct T {
         int mf(int);
         int mf(int,int);
    };
    __asm void  f(T*, int, int) {
        BL __cpp(static_cast<int (T::*)(int, int)>(&T::mf)) // calls T::mf(int,
int)
        BL __cpp(static_cast<void (*)(int)>(g)) // calls g(int)
        MOV pc, lr
    }
```

7．相关基类的关键字

利用以下关键字可以确定从对象起始处到其相关基类的偏移量：

```
__offsetof_base(D, B)
```

其中，B 必须是 D 的非虚拟基类。

该函数返回从 D 对象的起始处到其中 B 基子对象的起始处的偏移量。结果可能是零。必须将偏移量（以字节为单位）添加到 D* p 来执行。

static_cast<B*>(p) 的等效功能，如下程序段所示：

```
__asm B* my_static_base_cast(D* /*p*/) {
    if __offsetof_base(D, B) <> 0  //排除偏移量为 0 的情况
        ADD r0, r0, #__offsetof_base(D, B)
    endif
    MOV pc, lr
}
```

在汇编程序源代码中，这些关键字被转换为整数或逻辑常量。只能将它们用于__asm 函数，而不能用于__cpp 表达式。

8．成员函数类的关键字

以下关键字方便了从__asm 函数中调用虚拟或非虚拟成员函数。以__mcall 开头的关键字可用于虚拟和非虚拟函数。以__vcall 开头的关键字仅能用于虚拟函数。在调用静态成员函数的过程中，这些关键字没有特别的作用。

下面详细介绍这些关键字的使用。

① __mcall_is_virtual(D, f)

如果 f 是 D 中的虚拟成员函数或是 D 的基类，结果是{TRUE}，否则结果是{FALSE}。如果返回{TRUE}，可用虚拟调度进行调用，否则必须直接进行调用。

② __mcall_is_in_vbase(D, f)

如果 f 是 D 虚拟基类中的非静态成员函数，结果是{TRUE}，否则结果是{FALSE}。如果返回{TRUE}，必须用__mcall_offsetof_vbaseptr(D, f)进行 this 调整，否则必须用__mcall_this_offset(D, f)进行调整。

③ __mcall_this_offset(D, f)

其中 D 是类，f 是 D 中定义的非静态成员函数或是 D 的非虚拟基类。该函数返回从 D 对象的起始处到定义 f 的基的起始处的偏移量。在用指向 D 的指针调用 f 的过程中，这是必要的 this 调整。返回值在 D 中可找到 f 时为零，或者与__offsetof_base(D, B)相同，其中 B 为包含 f 的 D 非虚拟基类。在 D 的虚拟基类中找到 f 时，如果使用__mcall_this_offset(D, f)，则返回任意值，在程序中使用该返回值，汇编器将报告__mcall_this_offset 无效使用的错误。

④ __vcall_offsetof_vfunc(D, f)

其中 D 是类，f 是 D 中定义的虚拟函数或是 D 的基类。将偏移量返回到虚拟函数表，在该表中可以找到从虚拟函数表到虚拟函数的偏移量。在 f 不是虚拟成员函数时，如果使用__vcall_offsetof_vfunc(D, f)，则返回任意值，而在设计上使用该值时会导致汇编错误。

9．调用非静态成员函数

本小节列出了可以从 __asm 函数中调用虚拟或非虚拟函数的关键字。静态成员函数的参数不相同（没有 this），使得检测静态成员函数的关键字__mcall_is_static 不可用，因此调用位置很可能已经专用于调用静态成员函数。

（1）调用非虚拟成员函数

例如，在虚拟基（virtual base）或非虚拟基（non－virtual base）中，以下代码可用于调用虚拟函数：

```
// rp 包含指向 D 的指针，该程序的功能是实现在使用 rp 时调用 D 的非虚成员函数 f
// 所有参数准备好
// 假设并不返回一个结构类型
 if __mcall_is_in_vbase(D, f)
   ASSERT {FALSE}   // can't access virtual base
 else
   MOV r0, rp                              //使用指向 D 的指针 rp*
   ADD r0, r0, #__mcall_this_offset(D, f)  //地址调整
 endif
   BL __cpp(&D::f)
```

（2）调用虚拟成员函数

例如，在虚拟或非虚拟基中，以下代码可用于调用虚拟函数：

```
// rp 包含指向 D 的指针，该程序的功能是在使用 rp 时调用 D 的虚拟函数 f
// 所有参数准备好
// 假如函数并不返回一个结构类型
 if __mcall_is_in_vbase(D, f)
  ASSERT {FALSE}                              // 不能调用虚拟基
 else
  MOV r0, rp                                  // 使用指向 D 的指针 rp
  LDR r12, [rp]                               // 加载 vtable 表结构指针
  ADD r0, r0, #__mcall_this_offset(D, f)      // 地址调整
 endif
  MOV lr, pc                                  // 保存返回地址到 lr
  LDR pc, [r12, #__vcall_offsetof_vfunc(D, f)] // 调用函数 rp→f()
```

10．嵌入式汇编版本间的差异

不同版本的 ARM 编译器对嵌入式汇编程序的语法要求会有所差异。在具体使用时请参见相关文档。

值得注意的是，目前的嵌入式汇编器已经完全支持 ARMv6 指令集，也就是说可以在嵌入式汇编中使用 ARMv6 指令集中的指令。

12.1.3 内联汇编中使用 SP、LR 和 PC 寄存器的遗留问题

虽然目前的编译器不支持在内联汇编中使用 SP、LR 和 PC 寄存器，但在 RVCT v1.2 及其以前的编译器版本中是允许的。下面的例子显示了使用早期编译器版本，在内联汇编中使用 LR 寄存器的例子。

```
void func()
{
     int var;
     __asm
     {
         mov  var, lr  /* 得到func()函数的返回地址 */
     }
}
```

如果使用 RVCT v2.0 编译器编译上面的代码，编译器将报告以下错误。

```
Error:  #20: identifier "lr" is undefined
```

使用 RVCT v2.0 版本及其以后的编译器，要在 C 或 C++代码中使用汇编访问 SP、LR 和 PC 寄存器可以使用下面几种方法。

① 使用嵌入式汇编代码。嵌入式汇编支持所有的 ARM 指令，同时允许在代码中访问 SP、LR 和 PC 寄存器。

② 在内联汇编中使用以下一些指令。

- __current_pc()：访问 PC 寄存器。
- __current_sp()：访问 SP 寄存器。
- __return_address()：访问 LR，返回地址寄存器。

下面给出了两个访问 SP、LR 和 PC 寄存器的典型实例程序。

① 使用编译器给定的指令。

```
void printReg()
{
     unsigned int spReg, lrReg, pcReg;
     __asm {
         MOV spReg, __current_sp()
         MOV pcReg, __current_pc()
         MOV lrReg, __return_address()
     }
     printf("SP = 0x%X\n",spReg);
     printf("PC = 0x%X\n",pcReg);
     printf("LR = 0x%X\n",lrReg);
```

```
}
```

② 使用嵌入式汇编。

```
__asm void func()
{
    MOV r0, lr
    ...
    BX lr
}
```

使用嵌入式汇编可以使用调试器捕获程序的返回地址。

12.1.4 内联汇编代码与嵌入式汇编代码之间的差异

本节总结了内联汇编和嵌入式汇编在编译方法上存在的差异：

- 内联汇编代码使用高级处理器抽象，并在代码生成过程中与 C 和 C++代码集成。因此，编译程序将 C 和 C++代码与汇编代码一起进行优化。
- 与内联汇编代码不同，嵌入式汇编代码从 C 和 C++代码中分离出来单独进行汇编，产生与 C 和 C++源代码编译对象相结合的编译对象。
- 可通过编译程序来内联内联汇编代码，但无论是显式还是隐式，都无法内联嵌入式汇编代码。

表 12.1 总结了内联汇编程序与嵌入式汇编程序之间的主要差异。

表 12.1 内联汇编程序与嵌入式汇编程序之间的主要差异

功　能	嵌入式汇编程序	内联汇编程序
指令集	ARM 和 Thumb	仅支持 ARM
ARM 汇编指令伪操作	支持	不支持
ARMv6 指令集	支持	仅支持媒体指令
C/C++表达式	只支持常数表达式	完全支持
汇编代码是否优化	无优化	完全优化
能否被内联（Inling）	不可能	有可能被内联

续表

功　能	嵌入式汇编程序	内联汇编程序
寄存器访问	使用指定的物理寄存器，还可以使用 PC、LR 和 SP	使用虚拟寄存器。不能使用 PC、LR 和 SP 寄存器
是否自动产生返回指令	手工添加返回指令	指定产生（但不支持 BX、BXJ 和 BLX 指令）
是否支持 BKPT 指令	不直接支持	不支持

12.2 从汇编代码访问 C 全局变量

在汇编代码中访问 C 全局变量，只能通过地址间接访问全局变量。要访问全局变量，必须在汇编中使用 IMPORT 伪操作输入全局变量，然后将地址载入寄存器。可以根据变量的类型使用载入和存储指令访问该变量。

对于无符号变量，使用：

- LDRB/STRB：用于 char 型；
- LDRH/STRH：用于 short 型（对于 ARM 体系结构 v3，使用两个 LDRB/STRB 指令）；
- LDR/STR：用于 int 型。

对于有符号变量，请使用等效的有符号数的 Load/Store 指令，如 LDRSB 和 LDRSH。

对于少于 8 个字的小结构体可以用 LDM 和 STM 指令将其作为整体访问。同时也可以用适当类型的 Load/Store 指令访问结构的单个成员。为了访问成员，必须了解该成员地址相对于结构体开始处的偏移量。

下面的例子将整型全局变量 globvar 的地址载入 r1、将该地址中包含的值载入 r0、将它与 2 相加，然后将新值存回 globvar 中。

```
        PRESERVE8
        AREA    globals,CODE,READONLY
        EXPORT   asmsubroutine
        IMPORT   globvar
    asmsubroutine
        LDR  r1, =globvar   ;read address of globvar into
                    ;r1 from literal pool 从内存池中读取 globvar 变量的地址，加载到
r1 中
        LDR  r0, [r1]
        ADD  r0, r0, #2
        STR  r0, [r1]
        MOV  pc, lr
        END
```

12.3 在 C++中使用 C 头文件

本节描述如何在 C++代码中使用 C 头文件。从 C++调用 C 头文件之前，C 头文件必须包含在 extern "C"命令中。本节包含以下两部分内容：

- 在 C++中使用系统的 C 头文件；
- 在 C++中使用自定义的 C 头文件。

12.3.1 在 C++中使用系统 C 头文件

要包括标准的系统 C 头文件，如 stdio.h，不必进行任何特殊操作。只有使用 extern "C" 命令，由编译器自动包含标准 C 头文件。例如：

```
#include <stdio.h>
int main()
{
    ...     // C++ 代码
    return 0;
}
```

如果使用此语法包含头文件，则所有库名都放在全局命名空间中。

C++标准规定可以通过特定的 C++头文件获取 C 头文件。这些文件与标准 C 头文件一起安装在 install_directory\RVCT\Data\2.0\build_num \include \platform 目录下，可以用常规方法进行引用。例如：

```
#include <cstdio>
int main()
{
   ...     // C++ 代码
   return 0;
}
```

在 ARM C++中，这些头文件中包含（#include）C 头文件。如果使用此语法包含头文件，则所有 C++标准库名都在命名空间 std 中定义，包括 C 库名。这意味着必须使用下列方法之一来限定所有的库名称。

- 指定标准命名空间，例如：

```
std::printf("example\n");
```

- 使用 C++关键字“using”向全局命名空间输入一个名称：

```
using namespace std;
printf("example\n");
```

- 使用编译程序选项 --using_std。

12.3.2 在 C++中使用自定义的 C 头文件

要包含自己的 C 头文件，用户必须将#include 命令包在 extern "C"语句中。可以用以下方法完成此操作：

① 在#include 文件之前使用 extern，如下例所示。

```
// C++ code
```

```
extern "C" {
#include "my-header1.h"
#include "my-header2.h"
}
int main()
{
    // ...
    return 0;
}
```

- 将 extern "C"语句添加到头文件，如下例所示。

```
/* C header file */
#ifdef __cplusplus   /* Insert start of extern C construct */
extern "C" {
#endif
/* Body of header file */
#ifdef __cplusplus  /* Insert end of extern C construct */
}               /* The C header file can now be */
#endif           /* included in either C or C++ code. */
```

12.4 C、C++ 和 ARM 汇编语言之间的调用

本节提供一些示例，显示如何从 C++调用 C 和汇编语言代码，以及从 C 和汇编语言调用 C++ 代码。其中包括调用约定和数据类型。主要包括下面内容：

- 相互调用的一般规则；
- C++语言的特定信息；
- 调用示例。

只要遵循正确的过程调用标准 AAPCS，就可以混合调用 C、C++和汇编语言例程。有关 AAPCS 的更多信息，请参阅 ARM 相关文档。

12.4.1 相互调用的一般规则

以下一般规则适用于 C、C++和汇编语言之间的调用。有关的详细信息，请参阅 ARM 开发相关文档。

嵌入式汇编程序以及其与 ARM 嵌入式应用程序二进制接口（BSABI，Application Binary Interface for the ARM Architecture）的兼容使得混合语言编程更易于实现。它们可提供以下功能：

- 使用__cpp 关键字进行名称延伸；
- 传递隐含 this 参数的方式；

- 调用虚函数的方式；
- 引用的表示；
- 具有基类或虚成员函数的 C++类的类型布局；
- 非 POD（Plain Old Data）结构的类对象传递。

以下一般规则适用于混合语言编程：

- 使用 C 调用约定。
- 在 C++中，非成员函数可以声明为 extern "C"，以指定它们有 C 链接。带有 C 链接意味着定义函数的符号未延伸。C 链接可以用于以一种语言实现函数，然后用另一种语言调用它。
- 汇编语言模块所必须符合的 AAPCS 调用标准，应当适合于应用程序所使用的存储器模型。

以下规则适用于从 C 和汇编语言调用 C++函数：

- 要调用全局（非成员）C++函数，应将它声明为 extern "C"，以提供 C 链接。
- 成员函数（静态和非静态）总是有已延伸的名称。使用嵌入式汇编程序的__cpp 关键字，可以不必手工寻找已延伸的名称。
- 不能从 C 调用 C++内联函数，除非确保 C++编译器生成了函数的外联副本。例如，取得函数地址将导致生成外联副本。
- 非静态成员函数接受隐含 this 参数作为 r0 中的第一个自变量，或作为 r1 中第二个自变量（如果函数返回非 int 类结构）。静态成员函数不接受隐含 this 参数。

12.4.2 C++的特定信息

本节主要介绍一些专门适用于 C++的内容。

（1）C++调用约定

ARM C++使用与 ARM C 相同的调用约定，但在下面的情况下，调用规则有所不同：

- 调用非静态成员函数时，隐含的 this 参数是第一个自变量，或者是第二个自变量（如果被调用函数返回非 int 类的 struct）。这可能在将来的版本中有所变化。

（2）C++数据类型

ARM C++使用与 ARM C 相同的数据类型，但在以下几种情况下，情况有所不同：

- 如果 struct 或 class 类型的 C++对象没有基类或虚函数，则它们的布局与 ARM C 相同。如果这样的 struct 没有用户定义的复制赋值运算符或用户定义的析构函数，则它是 POD 结构。
- 引用表示为指针。
- C 函数指针和 C++（非成员）函数指针没有区别。

（3）符号名称延伸

链接程序将取消信息中符号名称的延伸。

在 C++程序中，C 名称必须声明为 extern "C"。ARM ISO C 头文件已经完成此操作。详细信息请参阅 ARM 相关文档。

12.4.3 混合编程调用举例

汇编程序、C 程序以及 C++程序相互调用时，要特别注意遵守相应的 AAPCS。下面一些

例子具体说明了在这些混合调用中应注意遵守的 AAPCS 规则。这些示例程序默认为使用非软件栈检查的 ATPCS 规则，因为它们执行栈操作时不检查栈溢出。

（1）从 C 调用汇编语言

下面的程序显示如何在 C 程序中调用汇编语言子程序，该段代码实现了将一个字符串复制到另一个字符串。

```
#include <stdio.h>
extern void strcopy(char *d, const char *s);
int main()
{   const char *srcstr = "First string - source ";
    char dststr[] = "Second string - destination ";
/* 下面将 dststr 作为数组进行操作 */
    printf("Before copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    strcopy(dststr,srcstr);
    printf("After copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    return (0);
}
```

下面为调用的汇编程序。

```
    PRESERVE8
    AREA    SCopy, CODE, READONLY
    EXPORT strcopy
Strcopy                     ;r0 指向目的字符串
                            ;r1 指向源字符串
    LDRB r2, [r1],#1        ;加载字节并更新源字符串指针地址
    STRB r2, [r0],#1        ;存储字节并更新目的字符串指针地址
    CMP r2, #0              ;判断是否为字符串结尾
    BNE strcopy             ;如果不是，程序跳转到 strcopy 继续拷贝
    MOV pc,lr               ;程序返回
    END
```

按以下步骤从命令行编译该示例：

① 输入 armasm -g scopy.s 编译汇编语言源代码。

② 输入 armcc -c -g strtest.c 编译 C 源代码。

③ 输入 armlink strtest.o scopy.o -o strtest 链接目标文件。

④ 将 ELF/DWARF2 兼容调试器与相应调试目标配合使用，运行映像。

（2）汇编语言调用 C 程序

下面的例子显示了如何从汇编语言调用 C 程序。

下面的子程序段定义了 C 语言函数。

```
int g(int a, int b, int c, int d, int e)
{
            return a + b + c + d + e;
}
```

下面的程序段显示了汇编语言调用。假设程序进入 f 时，r0 中的值为 i。

```
  ; int f(int i) { return g(i, 2*i, 3*i, 4*i, 5*i); }
  PRESERVE8
  EXPORT f
  AREA f, CODE, READONLY
  IMPORT g                      // 声明 C 程序 g()
  STR lr, [sp, #-4]!            // 保存返回地址 lr
  ADD r1, r0, r0                // 计算 2*i(第 2 个参数)
  ADD r2, r1, r0                // 计算 3*i(第 3 个参数)
  ADD r3, r1, r2                // 计算 5*i
  STR r3, [sp, #-4]!            // 第五个参数通过堆栈传递
  ADD r3, r1, r1                // 计算 4*i(第 4 个参数)
  BL g                          // 调用 C 程序
  ADD sp, sp, #4                // 从堆栈中删除第 5 个参数
  LDR pc, [sp], #4              // 返回
  END
```

（3）从 C++调用 C

下面的例子显示了如何从 C++程序中调用 C 函数。

下面的 C++程序调用了 C 程序。

```
struct S {                      // 本结构没有基类和虚函数

     S(int s):i(s) { }
     int i;
};
extern "C" void cfunc(S *);
// 被调用的 C 函数使用 extern “C” 声明
int f(){
      S s(2);                   // 初始化 's'
      cfunc(&s);                // 调用 C 函数 'cfunc' 将改变 's'
      return si*3;
}
```

下面显示了被调用的 C 程序代码。

```
struct S {
     int i;
};
void cfunc(struct S *p) {
/*定义被调用的 C 功能 */
     p->i += 5;
}
```

（4）从 C++中调用汇编

下面的例子显示了如何从 C++中调用汇编程序。

下面的例子为调用汇编程序的 C++代码。

```
struct S {                          // 本结果没有基类和虚拟函数
                                    //
     S(int s) : i(s) { }
     int i;
};
extern "C" void asmfunc(S *);       // 声明被调用的汇编函数

int f() {
      S s(2);                       // 初始化结构体 's'
      asmfunc(&s);                  // 调用汇编子程序 'asmfunc'

      return s.i * 3;
}
```

下面是被调用的汇编程序。

```
    PRESERVE8
    AREA Asm, CODE
    EXPORT asmfunc
asmfunc              // 被调用的汇编程序定义
    LDR r1, [r0]
    ADD r1, r1, #5
    STR r1, [r0]
    MOV pc, lr
    END
```

（5）从 C 中调用 C++

下面的例子显示了如何从 C++代码中调用 C 程序。

下面的代码显示了被调用 C++代码。

```
struct S {       // 本结构没有基类和虚拟函数
     S(int s) : i(s) { }
     int i;
};
extern "C" void cppfunc(S *p) {
// 定义被调用的 C++代码
// 连接了 C 功能
     p->i += 5;              //
}
```

调用了 C++代码的 C 函数。

```
struct S {
     int i;
};
extern void cppfunc(struct S *p);
/* 声明将会被调用的 C++功能 */
int f(void) {
      struct S s;
      s.i = 2;              /* 初始化 S */
      cppfunc(&s);          /* 调用 cppfunc 函数，该函数可能改变 S 的值 */
      return s.i * 3;
}
```

（6）从汇编中调用 C++程序

下面的代码显示了如何从汇编中调用 C++程序。

下面是被调用的 C++程序。

```
struct S {          // 本结构没有基类和虚拟函数
     S(int s) : i(s) { }
     int i;
};
extern "C" void cppfunc(S * p) {
// 定义被调用的 C++功能
// 功能函数体
     p->i += 5;
}
```

在汇编语言中，声明要调用的 C++功能，使用带连接的跳转指令调用 C++功能。

```
   AREA Asm, CODE
   IMPORT cppfunc                ;声明被调用的 C++ 函数名
```

```
    EXPORT  f
f
    STMFD  sp!,{lr}
    MOV    r0,#2
    STR    r0,[sp,#-4]!          ;初始化结构体
    MOV    r0,sp                 ;调用参数为指向结构体的指针
    BL     cppfunc               ;调用 C++功能'cppfunc'

    LDR    r0, [sp], #4
    ADD    r0, r0, r0,LSL #1
    LDMFD  sp!,{pc}
    END
```

（7）在 C 和 C++函数间传递参数

下面的例子显示了如何在 C 和 C++函数间传递参数。

下面的代码为 C++函数。

```
extern "C" int cfunc(const int&);
// 声明被调用的 C 函数
extern "C" int cppfunc(const int& r) {
// 定义将被 C 调用的 C++函数
     return 7 * r;
}
int f() {
     int i = 3;
     return cfunc(i);   // 相 C 函数传参
}
```

下面为 C 函数。

```
extern int cppfunc(const int*);
/* 声明将被调用的 C++函数 */
int cfunc(const int *p) {
/*定义被 C++调用的 C 函数*/
    int k = *p + 4;
    return cppfunc(&k);
}
```

（8）从 C 或汇编语言调用 C++

下面的例子综合显示了如何从 C 或汇编语言中调用非静态、非虚的 C++成员函数。可以使用编译器编译出的汇编程序查找已延伸的函数名。

下面是被调用的 C++成员函数。

```
struct T {
     T(int i) : t(i) { }
     int t;
     int f(int i);
};
int T::f(int i) { return i + t; }
// 定义将被 C 调用的 C++功能函数
extern "C" int cfunc(T*);
// 声明将被 C++调用的 C 函数
int f() {
     T t(5);                 // create an object of type T
     return cfunc(&t);
}
```

下面为调用 C++的 C 语言函数。

```
struct T;
extern int _ZN1T1fEi(struct T*, int);
     /* 被调用的 C++函数名 */
int cfunc(struct T* t) {
/* 定义被 C++调用的 C 函数 */
     return 3 * _ZN1T1fEi(t, 2);    /* 实现 3 乘以 t->f(2)功能 */
}
```

下面为调用 C++的汇编函数。

```
    EXPORT cfunc
    AREA foo, CODE
    IMPORT  _ZN1T1fEi
cfunc
    STMFD   sp!,{lr}            ;此时 r0 已经包含了指向对象的指针
    MOV r1, #2
    BL _ZN1T1fEi
    ADD r0, r0, r0, LSL #1      ;r0 乘以 3
    LDMFD sp!,{pc}
    END
```

下面的例子显示了如何用嵌入式汇编语言实现上面的例子。在此例中，使用 __cpp 关键字引用该函数。因此，用户不必了解已延伸的函数名。

```
struct T {
```

```
    T(int i) : t(i) { }
    int t;
    int f(int i);
};
int T::f(int i) { return i + t; }
// 定义被 C++调用的汇编功能
__asm int asm_func(T*) {
   STMFD sp!, {lr}
   MOV r1, #2;
   BL __cpp(T::f);
   ADD r0, r0, r0, LSL #1 ;r0 乘以 3
   LDMFD sp!, {pc}
}
int f() {
    T t(5); // 创建 T 类型的对象
    return asm_func(&t);
}
```

附：ARM 授权认证培训班：http://www.farsight.com.cn/courses/TS-ARM.htm